Lo

目錄

lodash

一个 JavaScript 的实用工具库, 表现一致性, 模块化, 高性能, 以及 可扩展

Example

```
_.assign({ 'a': 1 }, { 'b': 2 }, { 'c': 3 });
// → { 'a': 1, 'b': 2, 'c': 3 }

_.map([1, 2, 3], function(n) { return n * 3; });
// → [3, 6, 9]
```

特点

- ~100% 代码覆盖率
- 遵循 语义化版本控制规范
- 延迟计算链
- _(…) 支持隐式链
- _.ary & _.rearg 改变函数的实参个数和顺序
- _.at 更方便的获取数组或对象的值
- _.attempt 无需 try-catch 来处理可能会出错的执行函数
- _.before 与 _.after 互补
- _.bindKey 实现 “懒传参”
- _.chunk 按给定个数来拆分数组
- _.clone 支持对 `Date` & `RegExp` 对象的浅拷贝
- _.cloneDeep 深拷贝数组或对象
- _.curry & _.curryRight 用于创建 柯里化 函数
- _.debounce & _.throttle 处理函数防抖和节流
- _.defaultsDeep 深分配对象的可枚举属性
- _.fill 指定值填充数组
- _.findKey 按 keys 查找对象
- _.flow 与 _.flowRight (即 `_.compose`) 搭配
- _.forEach 支持提前中断
- _.forIn 遍历对象所有的可枚举属性
- _.forOwn 遍历对象的所有属性
- _.get & _.set 以 path 的方式获取和设置对象属性
- _.gt, _.gte, _.lt, & _.lte 关系比较方法
- _.inRange 检测给定的数值是否在指定范围内
- _.isNative 检测是否是原生函数
- _.isPlainObject & _.toPlainObject 检测是否是普通对象以及转换为普通对象
- _.isTypedArray 检测是否是类型数组
- _.mapKeys 按对象的 key 迭代，并返回新 key 的对象
- _.matches 支持深匹配对象

- _.matchesProperty & _.matches & _.property 互补
- _.merge 相当于递归版 _.extend
- _.method & _.methodOf 创建一个调用方法的函数
- _.modArgs 更高级的功能组合
- _.parseInt 兼容各环境
- _.pull, _.pullAt, & _.remove 方便调整数组
- _.random 支持返回浮点数
- _.restParam & _.spread 应用一个 rest arguments 和 Spread operator 参数传递给函数
- _.runInContext 无影响的 mixins 且更方便模拟
- _.slice 支持裁剪类数组
- _.sortByAll & _.sortByOrder 多个属性排序
- _.support 标记环境功能
- _.template 支持 *“imports”* 方式 & ES 字符串模板
- _.transform 更强大的 _.reduce 代替方法
- _.unzipWith & _.zipWith 指定如何重组分解后的数组
- _.valuesIn 获取所有可枚举属性的值
- _.xor & _.difference, _.intersection, & _.union 互补
- _.add, _.round, _.sum, 及更多 数学方法
- _.bind, _.curry, _.partial, & 及更多 支持自定参数占位
- _.capitalize, _.trim, & 及更多 string 方法
- _.clone, _.isEqual, & 及更多 接受自定回调函数
- _.dropWhile, _.takeWhile, & 及更多 互补 _.first, _.initial, _.last, & _.rest
- _.findLast, _.findLastKey, & 及更多 右结合方法
- _.includes, _.toArray, & 及更多 接受字符串方式
- _#commit & _#plant 配合链式队列
- _#thru 传递链式队列的值

关于翻译

- 该文档由 think2011 翻译，遵循 MIT协议， 定时保持与官方同步，翻译质量可能没法特别好，但会保证尽可能反复细心。
- 如果您有任何建议，或者意见，欢迎在此讨论，及时更正 ;-)。

入门指南

下载

查看 版本区别 来选择适合你的版本

- 现代版本 (压缩版) 适合现代浏览器（运行环境），例如：Chrome, Firefox, IE ≥ 9, & Safari ≥ 5.1
- 兼容版本 (压缩版) 同时兼容旧的浏览器（运行环境），例如：IE ≤ 8 & PhantomJS

引入

浏览器中使用:

```
<script src="lodash.js"></script>
```

AMD 规范中使用:

```
require(['lodash'], function(_) {});
```

使用 npm 安装:

```
$ {sudo -H} npm i -g npm$ npm i --save lodash
```

Node.js/io.js 中使用:

```
// 直接引用现代版本
var _ = require('lodash');

// 或引用某分类下的所有方法
var array = require('lodash/array');

// 或者引用具体方法 (很适合在 browserify/webpack 中做最小化打包)
var chunk = require('lodash/array/chunk');
```

查看 源码包 了解更多详情

注意: 在 REPL 中不要声明 特殊变量 "`_`"，安装 n_ 来代替。

模块格式

lodash 还有多种构建模块的格式

- npm 构建格式: 现代, 兼容, & 单个方法
- AMD 构建格式: 现代 & 兼容
- ES 构建格式: 现代

CDN 服务在 cdnjs & jsDelivr，通过 版本定制 构建你需要的模块，在找更多的功能用法? 试试 lodash-fp

深入了解

查看我们的 更新日志, 路线图, 以及 社区里的播客、文章、视频.

兼容性

在 Chrome 43-44, Firefox 38-39, IE 6-11, MS Edge, Safari 5-8, ChakraNode 0.12.2, Node.js 0.8.28, 0.10.40, 0.12.7, & 4.0.0, PhantomJS 1.9.8, RingoJS 0.11, & Rhino 1.7.6 测试通过

自动化测试 & 持续集成 已在运作，特别感谢 Sauce Labs 提供的浏览器自动化测试。

版本定制

通过版本定制可以很轻松的定制仅包含你所需功能的 lodash 版本 。更棒的是，我们已经帮你处理好了函数依赖和别名对应，查看 版本区别 & 选择一个适合你的版本。

使用 Grunt? 我们准备了 Grunt plugin 协助构建 lodash。

安装 lodash-cli 来作为 `lodash` 全局命令行工具:

```
$ {sudo -H} npm i -g npm
$ {sudo -H} npm i -g lodash-cli
$ lodash -h
```

注意: 请先卸载旧版本，再安装 `lodash-cli` 。

- 兼容版本构建，同时支持新旧运行环境，使用 `compat` 修饰。 *(默认)*

```
lodash compat
```

- 现代版本构建，针对新的环境，包括 ES5/ES6 支持，使用 `modern` 修饰。

```
lodash modern
```

- 严格模式版本构建, 开启 ES 严格模式，使用 `strict` 修饰。

```
lodash strict
```

- 模块化版本构建，拆散 lodash 为各个模块，使用 `modularize` 修饰。

```
lodash modularize
```

构建命令:

- 使用 `category` 命令以逗号分隔的方式传入需要的函数分类。可用的函数分类有： *"array", "chain", "collection", "date", "function", "lang", "object", "number", "string"*, & *"utility"*=。

```
lodash category=collection,function
```

- 使用 `exports` 命令以逗号分隔的方式传入导出 `lodash` 函数的方式，可用的方式有： *“amd”*, *“commonjs”*, *“es”*, *“global”*, *“iojs”*, *“node”*, *“npm”*, *“none”*, & *“umd”*.

```
lodash exports=amd,commonjs,iojs
```

- 使用 `iife` 命令指定代码替换 包裹 lodash 的 IIFE。

```
lodash iife="!function(window,undefined){%output%}(this)"
```

- 使用 `include` 命令以逗号分隔的方式传入需要包含的函数。

```
lodash include=each,filter,map
```

- 使用 `minus` 命令以逗号分隔的方式传入需要删减的函数/分类。

```
lodash modern minus=result,shuffle
```

- 使用 `plus` 命令以逗号分隔的方式传入需要补充的函数/分类。

```
lodash category=array plus=random,template
```

- 使用 `template` 命令传入路径相匹配的文件生成预编译模板。 注意: 预编译模板分配在 `_.templates` 对象上。

```
lodash template="./*.jst"
```

- 使用 `settings` 命令设置预编译模板时的模板语法。

```
lodash settings="{interpolate:/\{\{([\s\S]+?)\}\}/g}"
```

- 使用 `moduleId` 命令指定 lodash 的 AMD module ID 或 包含 lodash 的编译模板的 module ID。指定为 `none` 表示创建编译模板不依赖 lodash。

```
lodash moduleId=underscore
```

注意:

- 所有命令可以组合（除了 `compat` & `modern` ）

- `exports` 的值 *“es”* & *“npm”* 只能与 `modularize` 命令联用。
- `modularize` 命令使用最先的 `exports` 的值作为模块格式，忽略后续的值。
- 除非指定 `-o` 或 `--output` ，不然所有文件会保存在当前工作目录。
- Node.js 0.10.8-0.10.11 存在 bugs 导致无法最小化构建。

另外还支持以下选项：

```
-c, --stdout .......... 输出到stdout(译注：类似直接输出在终端那样)
-d, --development ..... 输出非最小化的开发代码
-h, --help ............ 显示帮助信息
-m, --source-map ...... 生成source-map文件
-o, --output .......... 指定输出的路径/文件名
-p, --production ...... 输出最小化后的产品代码
-s, --silent .......... 不显示常规的日志输出
-V, --version ......... 显示当前的 lodash 版本号
```

Array

chunk

- link
- source
- npm

```
_.chunk(array, [size=0])
```

将数组拆分成多个 size 长度的块，并组成一个新数组。 如果数组无法被分割成全部等长的块，那么最后剩余的元素将组成一个块。

参数

1. array (Array)

   需要被处理的数组

2. [size=0] (number)

   每个块的长度

返回值 (Array)

返回一个拆分好的新数组

示例

```
_.chunk(['a', 'b', 'c', 'd'], 2);
// => [['a', 'b'], ['c', 'd']]

_.chunk(['a', 'b', 'c', 'd'], 3);
// => [['a', 'b', 'c'], ['d']]
```

compact

- link
- source
- npm

```
_.compact(array)
```

创建一个移除了所有假值的数组。例如：`false`、`null`、`0`、`""`、`undefined`，以及`NaN`都是“假值”.

参数

1. array (Array)

   需要被处理的数组。

返回值 (Array)

返回移除了假值的数组。

示例

```
_.compact([0, 1, false, 2, '', 3]);
// => [1, 2, 3]
```

concat

- link
- source
- npm

```
_.concat(array, [values])
```

创建一个用任何数组 或 值连接的新数组。

参数

1. array (Array)

   需要被连接的数组

2. [values] (...*)

   需要被连接的值的队列

返回值 (Array)

返回连接后的新数组

示例

```
var array = [1];
var other = _.concat(array, 2, [3], [[4]]);

console.log(other);
// => [1, 2, 3, [4]]

console.log(array);
// => [1]
```

difference

- link
- source
- npm

```
_.difference(array, [values])
```

创建一个差异化后的数组，不包括使用 `SameValueZero` 方法提供的数组。

参数

1. array (Array)

   需要处理的数组

2. [values] (...Array)

   用于对比差异的数组

返回值 (Array)

返回一个差异化后的新数组

示例

```
_.difference([3, 2, 1], [4, 2]);
// => [3, 1]
```

differenceBy

- link
- source
- npm

```
_.differenceBy(array, [values], [iteratee=_.identity])
```

这个方法类似 `_.difference` ，除了它接受一个 iteratee 调用每一个数组和值。iteratee 会传入一个参数：(value)。

参数

1. array (Array)

   需要处理的数组

2. [values] (...Array)

   用于对比差异的数组

3. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回一个差异化后的新数组

示例

```
_.differenceBy([3.1, 2.2, 1.3], [4.4, 2.5], Math.floor);
// => [3.1, 1.3]

// 使用了 `_.property` 的回调结果
_.differenceBy([{ 'x': 2 }, { 'x': 1 }], [{ 'x': 1 }], 'x');
// => [{ 'x': 2 }]
```

differenceWith

- link
- source
- npm

```
_.differenceWith(array, [values], [comparator])
```

这个方法类似 `_.difference` ，除了它接受一个 comparator 调用每一个数组元素的值。 comparator 会传入2个参数：(arrVal, othVal)。

参数

1. array (Array)

   需要处理的数组

2. [values] (...Array)

   用于对比差异的数组

3. [comparator] (Function)

   这个函数会处理每一个元素

返回值 (Array)

返回一个差异化后的新数组

示例

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];

_.differenceWith(objects, [{ 'x': 1, 'y': 2 }], _.isEqual);
// => [{ 'x': 2, 'y': 1 }]
```

drop

- link
- source
- npm

```
_.drop(array, [n=1])
```

裁剪数组中的前 N 个数组，返回剩余的部分。

参数

1. array (Array)

   需要处理的数组

2. [n=1] (number)

   裁剪的个数

返回值 (Array)

返回数组的剩余的部分。

示例

```
_.drop([1, 2, 3]);
// => [2, 3]

_.drop([1, 2, 3], 2);
// => [3]

_.drop([1, 2, 3], 5);
// => []

_.drop([1, 2, 3], 0);
// => [1, 2, 3]
```

dropRight

- link
- source
- npm

```
_.dropRight(array, [n=1])
```

从右边开始裁剪数组中的 N 个数组，返回剩余的部分。

参数

1. array (Array)

   需要处理的数组

2. [n=1] (number)

   裁剪的个数

返回值 (Array)

返回数组的剩余的部分。

示例

```
_.dropRight([1, 2, 3]);
// => [1, 2]

_.dropRight([1, 2, 3], 2);
// => [1]

_.dropRight([1, 2, 3], 5);
// => []

_.dropRight([1, 2, 3], 0);
// => [1, 2, 3]
```

dropRightWhile

- link
- source
- npm

```
_.dropRightWhile(array, [predicate=_.identity])
```

从右边开始裁剪数组，起点从 `predicate` 返回假值开始。`predicate` 会传入3个参数：(value, index, array)。

参数

1. array (Array)

   需要处理的数组

2. [predicate=_.identity] (Function|Object|string)

   这个函数会在每一次迭代调用

返回值 (Array)

返回裁剪后的数组

示例

```
var resolve = _.partial(_.map, _, 'user');

var users = [
  { 'user': 'barney',  'active': true },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': false }
];

resolve( _.dropRightWhile(users, function(o) { return !o.active; })
// => ['barney']

// 使用了 `_.matches` 的回调结果
resolve( _.dropRightWhile(users, { 'user': 'pebbles', 'active': fal
// => ['barney', 'fred']

// 使用了 `_.matchesProperty` 的回调结果
resolve( _.dropRightWhile(users, ['active', false]) );
// => ['barney']

// 使用了 `_.property` 的回调结果
resolve( _.dropRightWhile(users, 'active') );
// => ['barney', 'fred', 'pebbles']
```

dropWhile

- link
- source
- npm

```
_.dropWhile(array, [predicate=_.identity])
```

裁剪数组，起点从 `predicate` 返回假值开始。`predicate` 会传入3个参数：(value, index, array)。

参数

1. array (Array)

   array 需要处理的数组

2. [predicate=_.identity] (Function|Object|string)

   这个函数会在每一次迭代调用

返回值 (Array)

Returns the slice of `array` .

示例

```
var users = [
  { 'user': 'barney',  'active': false },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': true }
];

_.dropWhile(users, function(o) { return !o.active; });
// => 结果: ['pebbles']

// 使用了 `_.matches` 的回调处理
_.dropWhile(users, { 'user': 'barney', 'active': false });
// => 结果: ['fred', 'pebbles']

// 使用了 `_.matchesProperty` 的回调处理
_.dropWhile(users, ['active', false]);
// => 结果: ['pebbles']

// 使用了 `_.property` 的回调处理
_.dropWhile(users, 'active');
// => 结果: ['barney', 'fred', 'pebbles']
```

fill

- link
- source
- npm

```
_.fill(array, value, [start=0], [end=array.length])
```

指定 `值` 填充数组，从 `start` 到 `end` 的位置，但不包括 `end` 本身的位置。

注意: 这个方法会改变数组

参数

1. array (Array)

   需要填充的数组

2. value (*)

   填充的值

3. [start=0] (number)

   开始位置

4. [end=array.length] (number)

   结束位置

返回值 (Array)

返回数组

示例

```
var array = [1, 2, 3];

_.fill(array, 'a');
console.log(array);
// => ['a', 'a', 'a']

_.fill(Array(3), 2);
// => [2, 2, 2]

_.fill([4, 6, 8, 10], '*', 1, 3);
// => [4, '*', '*', 10]
```

findIndex

- link
- source
- npm

```
_.findIndex(array, [predicate=_.identity])
```

这个方法类似 `_.find` 。除了它返回最先通过 `predicate` 判断为真值的元素的 index ，而不是元素本身。

参数

1. array (Array)

   需要搜索的数组

2. [predicate=_.identity] (Function|Object|string)

   这个函数会在每一次迭代调用

返回值 (number)

返回符合元素的 index，否则返回 `-1` 。

示例

```
var users = [
  { 'user': 'barney',  'active': false },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': true }
];

_.findIndex(users, function(o) { return o.user == 'barney'; });
// => 0

// 使用了 `_.matches` 的回调结果
_.findIndex(users, { 'user': 'fred', 'active': false });
// => 1

// 使用了 `_.matchesProperty` 的回调结果
_.findIndex(users, ['active', false]);
// => 0

// 使用了 `_.property` 的回调结果
_.findIndex(users, 'active');
// => 2
```

findLastIndex

- link
- source
- npm

```
_.findLastIndex(array, [predicate=_.identity])
```

这个方式类似 `_.findIndex` ，不过它是从右到左的。

参数

1. array (Array)

   需要搜索的数组

2. [predicate=_.identity] (Function|Object|string)

   这个函数会在每一次迭代调用

返回值 (number)

返回符合元素的 index，否则返回 `-1` 。

示例

```
var users = [
  { 'user': 'barney',  'active': true },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': false }
];

_.findLastIndex(users, function(o) { return o.user == 'pebbles'; }
// => 2

// 使用了 `_.matches` 的回调结果
_.findLastIndex(users, { 'user': 'barney', 'active': true });
// => 0

// 使用了 `_.matchesProperty` 的回调结果
_.findLastIndex(users, ['active', false]);
// => 2

// 使用了 `_.property` 的回调结果
_.findLastIndex(users, 'active');
// => 0
```

flatten

- link
- source
- npm

```
_.flatten(array)
```

向上一级展平数组嵌套

参数

1. array (Array)

   需要展平的数组

返回值 (Array)

返回展平后的新数组

示例

```
_.flatten([1, [2, [3, [4]], 5]]);
// => [1, 2, [3, [4]], 5]
```

flattenDeep

- link
- source
- npm

```
_.flattenDeep(array)
```

递归展平 `数组`.

参数

1. array (Array)

   需要展平的数组

返回值 (Array)

返回展平后的新数组

示例

```
_.flattenDeep([1, [2, [3, [4]], 5]]);
// => [1, 2, 3, 4, 5]
```

flattenDepth

- link
- source
- npm

```
_.flattenDepth(array, [depth=1])
```

根据 `depth` 递归展平 `数组` 的层级

参数

1. array (Array)

   需要展平的数组

2. [depth=1] (number)

   展平的层级

返回值 (Array)

返回展平后的新数组

示例

```
var array = [1, [2, [3, [4]], 5]];

_.flattenDepth(array, 1);
// => [1, 2, [3, [4]], 5]

_.flattenDepth(array, 2);
// => [1, 2, 3, [4], 5]
```

fromPairs

- link
- source
- npm

```
_.fromPairs(pairs)
```

反向版 `_.toPairs`，这个方法返回一个由键值对构成的对象。

参数

1. pairs (Array)

   键值对

返回值 (Object)

返回一个新对象

示例

```
_.fromPairs([['fred', 30], ['barney', 40]]);
// => { 'fred': 30, 'barney': 40 }
```

head first

- link
- source
- npm

```
_.head(array)
```

获得数组的首个元素

参数

1. array (Array)

   要检索的数组

返回值 (*)

返回数组中的首个元素

示例

```
_.head([1, 2, 3]);
// => 1

_.head([]);
// => undefined
```

indexOf

- link
- source
- npm

```
_.indexOf(array, value, [fromIndex=0])
```

根据 value 使用 SameValueZero 等值比较返回数组中首次匹配的 index， 如果 fromIndex 为负值，将从数组尾端索引进行匹配，如果将 fromIndex 设置为 true，将使用更快的二进制检索机制。

参数

1. array (Array)

   要检索的数组

2. value (*)

   要检索的值

3. [fromIndex=0] (number)

   需要检索的起始位置，如果为 true 将使用二进制检索方式。

返回值 (number)

返回匹配值的index，否则返回 -1。

示例

```
_.indexOf([1, 2, 1, 2], 2);
// => 1

// 使用了 `fromIndex`
_.indexOf([1, 2, 1, 2], 2, 2);
// => 3
```

initial

- link
- source
- npm

```
_.initial(需要检索的数组)
```

获取数组中除了最后一个元素之外的所有元素

参数

1. 需要检索的数组 (Array)

返回值 (Array)

返回没有最后一个元素的数组

示例

```
_.initial([1, 2, 3]);
// => [1, 2]
```

intersection

- link
- source
- npm

```
_.intersection([arrays])
```

创建一个包含所有使用 `SameValueZero` 进行等值比较后筛选的唯一值数组。

参数

1. [arrays] (...Array)

   需要处理的数组队列

返回值 (Array)

返回数组中所有数组共享元素的新数组

示例

```
_.intersection([2, 1], [4, 2], [1, 2]);
// => [2]
```

intersectionBy

- link
- source
- npm

```
_.intersectionBy([arrays], [iteratee=_.identity])
```

这个方法类似 `_.intersection`，除了它接受一个 iteratee 调用每一个数组和值。iteratee 会传入一个参数：(value)

参数

1. [arrays] (...Array)

   需要检索的数组

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回数组中共享元素的新数组

示例

```
_.intersectionBy([2.1, 1.2], [4.3, 2.4], Math.floor);
// => [2.1]

// 使用了 `_.property` 的回调结果
_.intersectionBy([{ 'x': 1 }], [{ 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 1 }]
```

intersectionWith

- link
- source
- npm

```
_.intersectionWith([arrays], [comparator])
```

这个方法类似 `_.intersection`，除了它接受一个 comparator 调用每一个数组和值。iteratee 会传入2个参数：((arrVal, othVal)

参数

1. [arrays] (...Array)

   需要检索的数组

2. [comparator] (Function)

   这个函数会处理每一个元素

返回值 (Array)

返回数组中共享元素的新数组

示例

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];
var others = [{ 'x': 1, 'y': 1 }, { 'x': 1, 'y': 2 }];

_.intersectionWith(objects, others, _.isEqual);
// => [{ 'x': 1, 'y': 2 }]
```

join

- link
- source
- npm

```
_.join(array, [separator=','])
```

将数组中的所有元素转换为由 `separator` 分隔的字符串。

参数

1. array (Array)

   需要转换的数组

2. [separator=','] (string)

   分隔符

返回值 (string)

返回连接好的字符串

示例

```
_.join(['a', 'b', 'c'], '~');
// => 'a~b~c'
```

last

- link
- source
- npm

```
_.last(array)
```

获取数组中的最后一个元素

参数

1. array (Array)

   要检索的数组

返回值 (*)

返回数组中的最后一个元素

示例

```
_.last([1, 2, 3]);
// => 3
```

lastIndexOf

- link
- source
- npm

```
_.lastIndexOf(array, value, [fromIndex=array.length-1])
```

这个方法类似 `_.indexOf` ，除了它是从右到左遍历元素的。 这个方法类似 `_.indexOf` except that it iterates over elements of `array` from right to left.

参数

1. array (Array)

   需要检索的数组

2. value (*)

   要检索的值

3. [fromIndex=array.length-1] (number)

   检索 index 的起点

返回值 (number)

返回匹配元素的 index，否则返回 -1

示例

```
_.lastIndexOf([1, 2, 1, 2], 2);
// => 3

// 使用了 `fromIndex`
_.lastIndexOf([1, 2, 1, 2], 2, 2);
// => 1
```

prototype.reverse

- link
- source
- npm

```
_.prototype.reverse()
```

反转数组，第一个元素移动到最后一位，第二个元素移动到倒数第二，类似这样。

注意: 这个方法会改变数组，根据 `Array#reverse`

返回值 (Array)

返回原数组

示例

```
var array = [1, 2, 3];

_.reverse(array);
// => [3, 2, 1]

console.log(array);
// => [3, 2, 1]
```

pull

- link
- source
- npm

```
_.pull(array, [values])
```

移除所有经过 `SameValueZero` 等值比较为 true 的元素

注意: 不同于 `_.without`，这个方法会改变数组。

参数

1. array (Array)

   需要调整的数组

2. [values] (...*)

   要移除的值

返回值 (Array)

返回数组本身

示例

```
var array = [1, 2, 3, 1, 2, 3];

_.pull(array, 2, 3);
console.log(array);
// => [1, 1]
```

pullAll

- link
- source
- npm

```
_.pullAll(array, values)
```

这个方式类似 `_.pull` ，除了它接受数组形式的一系列值。

注意: 不同于 `_.difference` ，这个方法会改变数组。

参数

1. array (Array)

   需要调整的数组

2. values (Array)

   要移除的值

返回值 (Array)

返回数组本身

示例

```
var array = [1, 2, 3, 1, 2, 3];

_.pullAll(array, [2, 3]);
console.log(array);
// => [1, 1]
```

pullAllBy

- link
- source
- npm

```
_.pullAllBy(array, values, [iteratee=_.identity])
```

这个方法类似 `_.pullAll`，除了它接受一个 comparator 调用每一个数组元素的值。 comparator 会传入一个参数：(value)。

注意: 不同于 `_.differenceBy`，这个方法会改变数组。

参数

1. array (Array)

   需要调整的数组

2. values (Array)

   要移除的值

3. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回数组本身

示例

```
var array = [{ 'x': 1 }, { 'x': 2 }, { 'x': 3 }, { 'x': 1 }];

_.pullAllBy(array, [{ 'x': 1 }, { 'x': 3 }], 'x');
console.log(array);
// => [{ 'x': 2 }]
```

pullAt

- link
- source
- npm

```
_.pullAt(array, [indexes])
```

根据给的 `indexes` 移除对应的数组元素并返回被移除的元素。

注意: 不同于 `_.at`，这个方法会改变数组。

参数

1. array (Array)

   需要调整的数组

2. [indexes] (...(number|number[])

   indexes 可以是特殊的数组队列，或者个别的单个或多个参数

返回值 (Array)

返回被移除的元素数组

示例

```
var array = [5, 10, 15, 20];
var evens = _.pullAt(array, 1, 3);

console.log(array);
// => [5, 15]

console.log(evens);
// => [10, 20]
```

remove

- link
- source
- npm

```
_.remove(array, [predicate=_.identity])
```

移除经过 `predicate` 处理为真值的元素，并返回被移除的元素。predicate 会传入3个参数：(value, index, array)

注意: Unlike `_.filter` ，这个方法会改变数组。

参数

1. array (Array)

   需要调整的数组

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回被移除的元素的数组

示例

```
var array = [1, 2, 3, 4];
var evens = _.remove(array, function(n) {
  return n % 2 == 0;
});

console.log(array);
// => [1, 3]

console.log(evens);
// => [2, 4]
```

slice

- link
- source
- npm

```
_.slice(array, [start=0], [end=array.length])
```

创建一个裁剪后的数组，从 start 到 end 的位置，但不包括 end 本身的位置。

注意: 这个方法用于代替 `Array#slice` 来确保数组正确返回

参数

1. array (Array)

   需要裁剪的数组

2. [start=0] (number)

   开始位置

3. [end=array.length] (number)

   结束位置

返回值 (Array)

返回裁剪后的数组

sortedIndex

- link
- source
- npm

```
_.sortedIndex(array, value)
```

使用二进制的方式检索来决定 value 应该插入在数组中位置。它的 index 应该尽可能的小以保证数组的排序。

参数

1. array (Array)

   需要检索的已排序数组

2. value (*)

   要评估位置的值

返回值 (number)

返回 value 应该在数组中插入的 index。

示例

```
_.sortedIndex([30, 50], 40);
// => 1

_.sortedIndex([4, 5], 4);
// => 0
```

sortedIndexBy

- link
- source
- npm

```
_.sortedIndexBy(array, value, [iteratee=_.identity])
```

这个方法类似 `_.sortedIndex`，除了它接受一个 iteratee 调用每一个数组和值来计算排序。iteratee 会传入一个参数：(value)。

参数

1. array (Array)

   需要检索的已排序数组

2. value (*)

   要评估位置的值

3. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (number)

返回 value 应该在数组中插入的 index。

示例

```
var dict = { 'thirty': 30, 'forty': 40, 'fifty': 50 };

_.sortedIndexBy(['thirty', 'fifty'], 'forty', _.propertyOf(dict));
// => 1

// 使用了 `_.property` 回调结果
_.sortedIndexBy([{ 'x': 4 }, { 'x': 5 }], { 'x': 4 }, 'x');
// => 0
```

sortedIndexOf

- link
- source
- npm

```
_.sortedIndexOf(array, value)
```

这个方法类似 `_.indexOf` ，除了它是执行二进制来检索已经排序的数组的。

参数

1. array (Array)

   需要检索的数组

2. value (*)

   要评估位置的值

返回值 (number)

返回匹配值的 index ，否则返回 `-1` .

示例

```
_.sortedIndexOf([1, 1, 2, 2], 2);
// => 2
```

sortedLastIndex

- link
- source
- npm

```
_.sortedLastIndex(array, value)
```

这个方法类似 `_.sortedIndex` ，除了它返回在 value 中尽可能大的 index 位置。

参数

1. array (Array)

   需要检索的已排序数组

2. value (*)

   要评估位置的值

返回值 (number)

返回 value 应该在数组中插入的 index。

示例

```
_.sortedLastIndex([4, 5], 4);
// => 1
```

sortedLastIndexBy

- link
- source
- npm

```
_.sortedLastIndexBy(array, value, [iteratee=_.identity])
```

这个方法类似 `_.sortedLastIndex`，除了它接受一个 iteratee 调用每一个数组和值来计算排序。iteratee 会传入一个参数：(value)。

参数

1. array (Array)

   需要检索的已排序数组

2. value (*)

   要评估位置的值

3. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (number)

返回 value 应该在数组中插入的 index。

示例

```
// 使用了 `_.property` 的回调结果
_.sortedLastIndexBy([{ 'x': 4 }, { 'x': 5 }], { 'x': 4 }, 'x');
// => 1
```

sortedLastIndexOf

- link
- source
- npm

```
_.sortedLastIndexOf(array, value)
```

这个方法类似 `_.lastIndexOf` ，除了它是执行二进制来检索已经排序的数组的。

参数

1. array (Array)

   需要检索的数组

2. value (*)

   要评估位置的值

返回值 (number)

返回匹配值的 index ，否则返回 -1.

示例

```
_.sortedLastIndexOf([1, 1, 2, 2], 2);
// => 3
```

sortedUniq

- link
- source
- npm

```
_.sortedUniq(array)
```

这个方法类似 `_.uniq`，除了它会排序并优化数组。

参数

1. array (Array)

   要调整的数组

返回值 (Array)

返回一个不重复的数组

示例

```
_.sortedUniq([1, 1, 2]);
// => [1, 2]
```

sortedUniqBy

- link
- source
- npm

```
_.sortedUniqBy(array, [iteratee])
```

这个方法类似 `_.uniqBy`，除了它接受一个 iteratee 调用每一个数组和值来排序并优化数组。

参数

1. array (Array)

   要调整的数组

2. [iteratee] (Function)

   这个函数会处理每一个元素

返回值 (Array)

返回一个不重复的数组

示例

```
_.sortedUniqBy([1.1, 1.2, 2.3, 2.4], Math.floor);
// => [1.1, 2.3]
```

tail

- link
- source
- npm

```
_.tail(array)
```

获取数组中除了第一个元素的剩余数组

参数

1. array (Array)

   要检索的数组

返回值 (Array)

返回数组中除了第一个元素的剩余数组

示例

```
_.tail([1, 2, 3]);
// => [2, 3]
```

take

- link
- source
- npm

```
_.take(array, [n=1])
```

从数组的起始元素开始提取 N 个元素。

参数

1. array (Array)

   需要处理的数组

2. [n=1] (number)

   要提取的个数

返回值 (Array)

返回提取的元素数组

示例

```
_.take([1, 2, 3]);
// => [1]

_.take([1, 2, 3], 2);
// => [1, 2]

_.take([1, 2, 3], 5);
// => [1, 2, 3]

_.take([1, 2, 3], 0);
// => []
```

takeRight

- link
- source
- npm

```
_.takeRight(array, [n=1])
```

从数组的结束元素开始提取 N 个数组

参数

1. array (Array)

   需要处理的数组

2. [n=1] (number)

   要提取的个数

返回值 (Array)

返回提取的元素数组

示例

```
_.takeRight([1, 2, 3]);
// => [3]

_.takeRight([1, 2, 3], 2);
// => [2, 3]

_.takeRight([1, 2, 3], 5);
// => [1, 2, 3]

_.takeRight([1, 2, 3], 0);
// => []
```

takeRightWhile

- link
- source
- npm

```
_.takeRightWhile(array, [predicate=_.identity])
```

从数组的最右边开始提取数组，直到 `predicate` 返回假值。`predicate` 会传入三个参数：(value, index, array)。

参数

1. array (Array)

   需要处理的数组

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回提取的元素数组

示例

```
var users = [
  { 'user': 'barney',  'active': true },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': false }
];

_.takeRightWhile(users, function(o) { return !o.active; });
// => 结果:  ['fred', 'pebbles']

// 使用了 `_.matches` 的回调处理
_.takeRightWhile(users, { 'user': 'pebbles', 'active': false });
// => 结果:  ['pebbles']

// 使用了 `_.matchesProperty` 的回调处理
_.takeRightWhile(users, ['active', false]);
// => 结果:  ['fred', 'pebbles']

// 使用了 `_.property` 的回调处理
_.takeRightWhile(users, 'active');
// => []
```

takeWhile

- link
- source
- npm

```
_.takeWhile(array, [predicate=_.identity])
```

从数组的开始提取数组，直到 predicate 返回假值。predicate 会传入三个参数：(value, index, array)。

参数

1. array (Array)

   需要处理的数组

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回提取的元素数组

示例

```
var users = [
  { 'user': 'barney',  'active': false },
  { 'user': 'fred',    'active': false},
  { 'user': 'pebbles', 'active': true }
];

_.takeWhile(users, function(o) { return !o.active; });
// => objects for ['barney', 'fred']

// 使用了 `_.matches` 的回调处理
_.takeWhile(users, { 'user': 'barney', 'active': false });
// =>结果: ['barney']

// 使用了 `_.matchesProperty` 的回调处理
_.takeWhile(users, ['active', false]);
// =>结果: ['barney', 'fred']

// 使用了 `_.property` 的回调处理
_.takeWhile(users, 'active');
// => []
```

union

- link
- source
- npm

```
_.union([arrays])
```

创建顺序排列的唯一值组成的数组。所有值经过 `SameValueZero` 等值比较。

参数

1. [arrays] (...Array)

   需要处理的数组队列

返回值 (Array)

返回处理好的数组

示例

```
_.union([2, 1], [4, 2], [1, 2]);
// => [2, 1, 4]
```

unionBy

- link
- source
- npm

```
_.unionBy([arrays], [iteratee=_.identity])
```

这个方法类似 `_.union` ，除了它接受一个 iteratee 调用每一个数组和值。iteratee 会传入一个参数：(value)。

参数

1. [arrays] (...Array)

   需要处理的数组队列

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回处理好的数组

示例

```
_.unionBy([2.1, 1.2], [4.3, 2.4], Math.floor);
// => [2.1, 1.2, 4.3]

// 使用了 `_.property` 的回调结果
_.unionBy([{ 'x': 1 }], [{ 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 1 }, { 'x': 2 }]
```

unionWith

- link
- source
- npm

```
_.unionWith([arrays], [comparator])
```

这个方法类似 `_.union` ，除了它接受一个 comparator 调用每一个数组元素的值。 comparator 会传入2个参数：(arrVal, othVal)。

参数

1. [arrays] (...Array)

   需要处理的数组队列

2. [comparator] (Function)

   这个函数会处理每一个元素

返回值 (Array)

返回处理好的数组

示例

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];
var others = [{ 'x': 1, 'y': 1 }, { 'x': 1, 'y': 2 }];

_.unionWith(objects, others, _.isEqual);
// => [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }, { 'x': 1, 'y': 1 }]
```

uniq

- link
- source
- npm

```
_.uniq(array)
```

创建一个不重复的数组副本。使用了 `SameValueZero` 等值比较。只有首次出现的元素才会被保留。

参数

1. array (Array)

   需要处理的数组

返回值 (Array)

返回不重复的数组

示例

```
_.uniq([2, 1, 2]);
// => [2, 1]
```

uniqBy

- link
- source
- npm

```
_.uniqBy(array, [iteratee=_.identity])
```

这个方法类似 `_.uniq` ，除了它接受一个 iteratee 调用每一个数组和值来计算唯一性。iteratee 会传入一个参数：(value)。

参数

1. array (Array)

   需要处理的数组

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回不重复的数组

示例

```
_.uniqBy([2.1, 1.2, 2.3], Math.floor);
// => [2.1, 1.2]

// 使用了 `_.property` 的回调结果
_.uniqBy([{ 'x': 1 }, { 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 1 }, { 'x': 2 }]
```

uniqWith

- link
- source
- npm

```
_.uniqWith(array, [comparator])
```

这个方法类似 `_.uniq`，除了它接受一个 `comparator` 来比较计算唯一性。`comparator` 会传入2个参数：(arrVal, othVal)

参数

1. array (Array)

   需要处理的数组
2. [comparator] (Function)

   这个函数会处理每一个元素

返回值 (Array)

返回不重复的数组

示例

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 },  { 'x': 1,

_.uniqWith(objects, _.isEqual);
// => [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }]
```

unzip

- link
- source
- npm

```
_.unzip(array)
```

这个方法类似 `_.zip`，除了它接收一个打包后的数组并且还原为打包前的状态。

参数

1. array (Array)

   需要解包的已打包数组

返回值 (Array)

返回一个解包后的数组

示例

```
var zipped = _.zip(['fred', 'barney'], [30, 40], [true, false]);
// => [['fred', 30, true], ['barney', 40, false]]

_.unzip(zipped);
// => [['fred', 'barney'], [30, 40], [true, false]]
```

unzipWith

- link
- source
- npm

```
_.unzipWith(array, [iteratee=_.identity])
```

这个方法类似 `_.unzip` ，除了它接受一个 iteratee 来决定如何重组解包后的数组。iteratee 会传入4个参数：(accumulator, value, index, group)。每组的第一个元素作为初始化的值

参数

1. array (Array)

   需要解包的已打包数组

2. [iteratee=_.identity] (Function)

   决定如何重组解包后的元素

返回值 (Array)

返回一个解包后的数组

示例

```
var zipped = _.zip([1, 2], [10, 20], [100, 200]);
// => [[1, 10, 100], [2, 20, 200]]

_.unzipWith(zipped, _.add);
// => [3, 30, 300]
```

without

- link
- source
- npm

```
_.without(array, [values])
```

创建一个移除了所有提供的 values 的数组。使用了 `SameValueZero` 等值比较。

参数

1. array (Array)

   要处理的数组

2. [values] (...*)

   要排除的值

返回值 (Array)

返回一个处理好的新数组

示例

```
_.without([1, 2, 1, 3], 1, 2);
// => [3]
```

xor

- link
- source
- npm

```
_.xor([arrays])
```

创建一个包含了所有唯一值的数组。使用了 symmetric difference 等值比较。

参数

1. [arrays] (...Array)

   要处理的数组

返回值 (Array)

包含了所有唯一值的新数组

示例

```
_.xor([2, 1], [4, 2]);
// => [1, 4]
```

xorBy

- link
- source
- npm

```
_.xorBy([arrays], [iteratee=_.identity])
```

这个方法类似 `_.xor` ，除了它接受一个 iteratee 调用每一个数组和值。iteratee 会传入一个参数：(value)。

参数

1. [arrays] (...Array)

   要处理的数组

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

包含了所有唯一值的新数组

示例

```
_.xorBy([2.1, 1.2], [4.3, 2.4], Math.floor);
// => [1.2, 4.3]

// 使用了 `_.property` 的回调结果
_.xorBy([{ 'x': 1 }], [{ 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 2 }]
```

xorWith

- link
- source
- npm

```
_.xorWith([arrays], [comparator])
```

这个方法类似 `_.xor` ，除了它接受一个 comparator 调用每一个数组元素的值。comparator 会传入2个参数：(arrVal, othVal)。

参数

1. [arrays] (...Array)

   要处理的数组

2. [comparator] (Function)

   这个函数会处理每一个元素

返回值 (Array)

包含了所有唯一值的新数组

示例

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];
var others = [{ 'x': 1, 'y': 1 }, { 'x': 1, 'y': 2 }];

_.xorWith(objects, others, _.isEqual);
// => [{ 'x': 2, 'y': 1 }, { 'x': 1, 'y': 1 }]
```

zip

- link
- source
- npm

```
_.zip([arrays])
```

创建一个打包所有元素后的数组。第一个元素包含所有提供数组的第一个元素，第二个包含所有提供数组的第二个元素，以此类推。

参数

1. [arrays] (...Array)

   要处理的数组队列

返回值 (Array)

返回一个打包后的数组

示例

```
_.zip(['fred', 'barney'], [30, 40], [true, false]);
// => [['fred', 30, true], ['barney', 40, false]]
```

zipObject

- link
- source
- npm

```
_.zipObject([props=[]], [values=[]])
```

这个方法类似 `_.fromPairs` ，除了它接受2个数组，一个作为属性名，一个作为属性值。

参数

1. [props=[]] (Array)

   属性名

2. [values=[]] (Array)

   属性值

返回值 (Object)

返回一个新的对象

示例

```
_.zipObject(['a', 'b'], [1, 2]);
// => { 'a': 1, 'b': 2 }
```

zipObjectDeep

- link
- source
- npm

```
_.zipObjectDeep([props=[]], [values=[]])
```

这个方法类似 `_.zipObject` ，除了它支持属性路径。 This method is like `_.zipObject` except that it supports property paths.

参数

1. [props=[]] (Array)

   属性名

2. [values=[]] (Array)

   属性值

返回值 (Object)

返回新的对象

示例

```
_.zipObjectDeep(['a.b[0].c', 'a.b[1].d'], [1, 2]);
// => { 'a': { 'b': [{ 'c': 1 }, { 'd': 2 }] } }
```

zipWith

- link
- source
- npm

```
_.zipWith([arrays])
```

这个方法类似 _.zip，除了它接受一个 iteratee 决定如何重组值。 iteratee 会调用每一组元素。

参数

1. [arrays] (...Array)

   要处理的数组队列

返回值 (Array)

返回一个打包后的数组

示例

```
_.zipWith([1, 2], [10, 20], [100, 200], function(a, b, c) {
  return a + b + c;
});
// => [111, 222]
```

Collection

countBy

- link
- source
- npm

```
_.countBy(collection, [iteratee=_.identity])
```

创建一个组成对象，key是经过 `iteratee` 处理的集合的结果，value 是处理结果的次数。 `iteratee` 会传入一个参数：(value)。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Object)

返回一个组成汇总的对象

示例

```
_.countBy([6.1, 4.2, 6.3], Math.floor);
// => { '4': 1, '6': 2 }

_.countBy(['one', 'two', 'three'], 'length');
// => { '3': 2, '5': 1 }
```

every

- link
- source
- npm

```
_.every(collection, [predicate=_.identity])
```

通过 `predicate` 检查集合中的元素是否都返回 真值，只要 `predicate` 返回一次假值，遍历就停止，并返回 false。 `predicate` 会传入3个参数：(value, index|key, collection)

参数

1. collection (Array|Object)

   需要遍历的集合

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (boolean)

返回 true，如果所有元素经 predicate 检查都为真值，否则返回 false。

示例

```
_.every([true, 1, null, 'yes'], Boolean);
// => false

var users = [
  { 'user': 'barney', 'active': false },
  { 'user': 'fred',   'active': false }
];

// 使用了 `_.matches` 的回调结果
_.every(users, { 'user': 'barney', 'active': false });
// => false

// 使用了 `_.matchesProperty` 的回调结果
_.every(users, ['active', false]);
// => true

// 使用了 `_.property` 的回调结果
_.every(users, 'active');
// => false
```

filter

- link
- source
- npm

```
_.filter(collection, [predicate=_.identity])
```

遍历集合中的元素，筛选出一个经过 `predicate` 检查结果为真值的数组，predicate 会传入3个参数：(value, index|key, collection)。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回筛选结果的新数组

示例

```
var resolve = _.partial(_.map, _, 'user');

var users = [
  { 'user': 'barney', 'age': 36, 'active': true },
  { 'user': 'fred',   'age': 40, 'active': false }
];

resolve( _.filter(users, function(o) { return !o.active; }) );
// => ['fred']

// 使用了 `_.matches` 的回调结果
resolve( _.filter(users, { 'age': 36, 'active': true }) );
// => ['barney']

// 使用了 `_.matchesProperty` 的回调结果
resolve( _.filter(users, ['active', false]) );
// => ['fred']

// 使用了 `_.property` 的回调结果
resolve( _.filter(users, 'active') );
// => ['barney']
```

find

- link
- source
- npm

```
_.find(collection, [predicate=_.identity])
```

遍历集合中的元素，返回最先经 `predicate` 检查为真值的元素。 predicate 会传入3个元素：(value, index|key, collection)。

参数

1. collection (Array|Object)

   要检索的集合

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (*)

返回匹配元素，否则返回 `undefined`

示例

```
var users = [
  { 'user': 'barney',  'age': 36, 'active': true },
  { 'user': 'fred',    'age': 40, 'active': false },
  { 'user': 'pebbles', 'age': 1,  'active': true }
];

_.find(users, function(o) { return o.age < 40; });
// => 结果: 'barney'

// 使用了 `_.matches` 的回调结果
_.find(users, { 'age': 1, 'active': true });
// => 结果: 'pebbles'

// 使用了 `_.matchesProperty` 的回调结果
_.find(users, ['active', false]);
// => 结果: 'fred'

// 使用了 `_.property` 的回调结果
_.find(users, 'active');
// => 结果: 'barney'
```

findLast

- link
- source
- npm

```
_.findLast(collection, [predicate=_.identity])
```

这个方法类似 `_.find` ，除了它是从右至左遍历集合的。

参数

1. collection (Array|Object)

   要检索的集合

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (*)

返回匹配元素，否则返回 `undefined`

示例

```
_.findLast([1, 2, 3, 4], function(n) {
  return n % 2 == 1;
});
// => 3
```

flatMap

- link
- source
- npm

```
_.flatMap(collection, [iteratee=_.identity])
```

创建一个扁平化的数组，每一个值会传入 iteratee 处理，处理结果会与值合并。iteratee 会传入3个参数：(value, index|key, array)。

参数

1. collection (Array|Object)

   需要遍历的数组

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会在每一次迭代调用

返回值 (Array)

返回新数组

示例

```
function duplicate(n) {
  return [n, n];
}

_.flatMap([1, 2], duplicate);
// => [1, 1, 2, 2]
```

forEach each

- link
- source
- npm

```
_.forEach(collection, [iteratee=_.identity])
```

调用 iteratee 遍历集合中的元素， iteratee 会传入3个参数：(value, index|key, collection)。 如果显式的返回 false ，iteratee 会提前退出。

注意: 与其他集合方法一样，对象的 length 属性也会被遍历，避免这种情况，可以用 *.forIn* 或者 .forOwn 代替。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

返回值 (Array|Object)

返回集合

示例

```
_([1, 2]).forEach(function(value) {
  console.log(value);
});
// => 输出 `1` 和 `2`

_.forEach({ 'a': 1, 'b': 2 }, function(value, key) {
  console.log(key);
});
// => 输出 'a' 和 'b' (不保证遍历的顺序)
```

forEachRight eachRight

- link
- source
- npm

```
_.forEachRight(collection, [iteratee=_.identity])
```

这个方法类似 `_.forEach`，除了它是从右到左遍历的集合中的元素的。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

返回值 (Array|Object)

返回集合

示例

```
_.forEachRight([1, 2], function(value) {
  console.log(value);
});
// => 输出 `2` 和 `1`
```

groupBy

- link
- source
- npm

```
_.groupBy(collection, [iteratee=_.identity])
```

创建一个对象组成，key 是经 iteratee 处理的结果， value 是产生 key 的元素数组。 iteratee 会传入1个参数：(value)。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Object)

返回一个组成汇总的对象

示例

```
_.groupBy([6.1, 4.2, 6.3], Math.floor);
// => { '4': [4.2], '6': [6.1, 6.3] }

// 使用了 `_.property` 的回调结果
_.groupBy(['one', 'two', 'three'], 'length');
// => { '3': ['one', 'two'], '5': ['three'] }
```

includes

- link
- source
- npm

```
_.includes(collection, value, [fromIndex=0])
```

检查 值 是否在 集合中，如果集合是字符串，那么检查 值 是否在字符串中。 其他情况用 `SameValueZero` 等值比较。 如果指定 `fromIndex` 是负数，从结尾开始检索。

参数

1. collection (Array|Object|string)

   要检索的集合

2. value (*)

   要检索的值

3. [fromIndex=0] (number)

   要检索的 index 位置

返回值 (boolean)

如果找到 value 返回 ture， 否则返回 false。

示例

```
_.includes([1, 2, 3], 1);
// => true

_.includes([1, 2, 3], 1, 2);
// => false

_.includes({ 'user': 'fred', 'age': 40 }, 'fred');
// => true

_.includes('pebbles', 'eb');
// => true
```

invokeMap

- link
- source
- npm

```
_.invokeMap(collection, path, [args])
```

调用 `path` 的方法处理集合中的每一个元素，返回处理的数组。 如何附加的参数会传入到调用方法中。如果方法名是个函数，集合中的每个元素都会被调用到。

参数

1. collection (Array|Object)

   需要遍历的集合

2. path (Array|Function|string)

   要调用的方法名 或者 这个函数会处理每一个元素

3. [args] (...*)

   The arguments to invoke each method with.

返回值 (Array)

返回数组结果

示例

```
_.invokeMap([[5, 1, 7], [3, 2, 1]], 'sort');
// => [[1, 5, 7], [1, 2, 3]]

_.invokeMap([123, 456], String.prototype.split, '');
// => [['1', '2', '3'], ['4', '5', '6']]
```

keyBy

- link
- source
- npm

```
_.keyBy(collection, [iteratee=_.identity])
```

创建一个对象组成。key 是经 `iteratee` 处理的结果，value 是产生key的元素。iteratee 会传入1个参数：(value)。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Object)

返回一个组成汇总的对象

示例

```
var array = [
  { 'dir': 'left', 'code': 97 },
  { 'dir': 'right', 'code': 100 }
];

_.keyBy(array, function(o) {
  return String.fromCharCode(o.code);
});
// => { 'a': { 'dir': 'left', 'code': 97 }, 'd': { 'dir': 'right',

_.keyBy(array, 'dir');
// => { 'left': { 'dir': 'left', 'code': 97 }, 'right': { 'dir': 'r
```

map

- link
- source
- npm

```
_.map(collection, [iteratee=_.identity])
```

创建一个经过 `iteratee` 处理的集合中每一个元素的结果数组。 iteratee 会传入3个参数：(value, index|key, collection)。

有许多 lodash 的方法以 iteratees 的身份守护其工作，例如： `_.every` , `_.filter` , `_.map` , `_.mapValues` , `_.reject` , 以及 `_.some`

被守护的有: `ary` , `curry` , `curryRight` , `drop` , `dropRight` , `every` , `fill` , `invert` , `parseInt` , `random` , `range` , `rangeRight` , `slice` , `some` , `sortBy` , `take` , `takeRight` , `template` , `trim` , `trimEnd` , `trimStart` , 以及 `words`

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回映射后的新数组

示例

```
function square(n) {
  return n * n;
}

_.map([4, 8], square);
// => [16, 64]

_.map({ 'a': 4, 'b': 8 }, square);
// => [16, 64] (无法保证遍历的顺序)

var users = [
  { 'user': 'barney' },
  { 'user': 'fred' }
];

// 使用了 `_.property` 的回调结果
_.map(users, 'user');
// => ['barney', 'fred']
```

orderBy

- link
- source
- npm

```
_.orderBy(collection, [iteratees=[_.identity]], [orders])
```

这个方法类似 `_.sortBy`，除了它允许指定 iteratees 结果如何排序。 如果没指定 `orders`，所有值以升序排序。 其他情况，指定 "desc" 降序，指定 "asc" 升序其对应值。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratees=[_.identity]] (Function[]|Object[]|string[])

   通过 iteratees 决定排序

3. [orders] (string[])

   决定 iteratees 的排序方法

返回值 (Array)

排序排序后的新数组

示例

```
var users = [
  { 'user': 'fred',   'age': 48 },
  { 'user': 'barney', 'age': 34 },
  { 'user': 'fred',   'age': 42 },
  { 'user': 'barney', 'age': 36 }
];

// 以 `user` 升序排序 再 以 `age` 降序排序。
_.orderBy(users, ['user', 'age'], ['asc', 'desc']);
// => 结果: [['barney', 36], ['barney', 34], ['fred', 48], ['fred',
```

partition

- link
- source
- npm

```
_.partition(collection, [predicate=_.identity])
```

创建一个拆分为两部分的数组。 第一部分是 `predicate` 检查为真值的，第二部分是 `predicate` 检查为假值的。 predicate 会传入3个参数：(value, index|key, collection)。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回分组元素的数组

示例

```
var users = [
  { 'user': 'barney',  'age': 36, 'active': false },
  { 'user': 'fred',    'age': 40, 'active': true },
  { 'user': 'pebbles', 'age': 1,  'active': false }
];

_.partition(users, function(o) { return o.active; });
// => 结果: [['fred'], ['barney', 'pebbles']]

// 使用了 `_.matches` 的回调结果
_.partition(users, { 'age': 1, 'active': false });
// => 结果: [['pebbles'], ['barney', 'fred']]

// 使用了 `_.matchesProperty` 的回调结果
_.partition(users, ['active', false]);
// => 结果: [['barney', 'pebbles'], ['fred']]

// 使用了 `_.property` 的回调结果
_.partition(users, 'active');
// => 结果: [['fred'], ['barney', 'pebbles']]
```

reduce

- link
- source
- npm

```
_.reduce(collection, [iteratee=_.identity], [accumulator])
```

通过 `iteratee` 遍历集合中的每个元素。 每次返回的值会作为下一次 `iteratee` 使用。 如果没有提供 `accumulator` ，则集合中的第一个元素作为 `accumulator` 。 iteratee 会传入4个参数：(accumulator, value, index|key, collection)。

有许多 lodash 的方法以 iteratees 的身份守护其工作，例如： `_.reduce` , `_.reduceRight` , 以及 `_.transform` .

被守护的有: `assign` , `defaults` , `defaultsDeep` , `includes` , `merge` , `orderBy` , 以及 `sortBy`

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

3. [accumulator] (*)

   初始化的值

返回值 (*)

返回累加后的值

示例

```
_.reduce([1, 2], function(sum, n) {
  return sum + n;
}, 0);
// => 3

_.reduce({ 'a': 1, 'b': 2, 'c': 1 }, function(result, value, key) 
  (result[value] || (result[value] = [])).push(key);
  return result;
}, {});
// => { '1': ['a', 'c'], '2': ['b'] } (无法保证遍历的顺序)
```

reduceRight

- link
- source
- npm

```
_.reduceRight(collection, [iteratee=_.identity], [accumulator])
```

这个方法类似 `_.reduce` ，除了它是从右到左遍历的。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

3. [accumulator] (*)

   初始化的值

返回值 (*)

返回累加后的值

示例

```
var array = [[0, 1], [2, 3], [4, 5]];

_.reduceRight(array, function(flattened, other) {
  return flattened.concat(other);
}, []);
// => [4, 5, 2, 3, 0, 1]
```

reject

- link
- source
- npm

```
_.reject(collection, [predicate=_.identity])
```

反向版 `_.filter`，这个方法返回 `predicate` 检查为非真值的元素。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Array)

返回过滤后的新数组

示例

```
var users = [
  { 'user': 'barney', 'age': 36, 'active': false },
  { 'user': 'fred',   'age': 40, 'active': true }
];

_.reject(users, function(o) { return !o.active; });
// => 结果: ['fred']

// 使用了 `_.matches` 的回调结果
_.reject(users, { 'age': 40, 'active': true });
// => 结果: ['barney']

// 使用了 `_.matchesProperty` 的回调结果
_.reject(users, ['active', false]);
// => 结果: ['fred']

// 使用了 `_.property` 的回调结果
_.reject(users, 'active');
// => 结果: ['barney']
```

sample

- link
- source
- npm

```
_.sample(collection)
```

从集合中随机获得元素

参数

1. collection (Array|Object)

   要取样的集合

返回值 (*)

返回随机元素

示例

```
_.sample([1, 2, 3, 4]);
// => 2
```

sampleSize

- link
- source
- npm

```
_.sampleSize(collection, [n=0])
```

获得从集合中随机获得 `N` 个元素 Gets `n` random elements from `collection` .

参数

1. collection (Array|Object)

   要取样的集合

2. [n=0] (number)

   要取得的元素个数

返回值 (Array)

返回随机元素

示例

```
_.sampleSize([1, 2, 3], 2);
// => [3, 1]

_.sampleSize([1, 2, 3], 4);
// => [2, 3, 1]
```

shuffle

- link
- source
- npm

```
_.shuffle(collection)
```

创建一个被打乱元素的集合。 使用了 Fisher-Yates shuffle 版本。

参数

1. collection (Array|Object)

   要打乱的集合

返回值 (Array)

返回一个被打乱元素的新集合

示例

```
_.shuffle([1, 2, 3, 4]);
// => [4, 1, 3, 2]
```

size

* link
* source
* npm

```
_.size(collection)
```

返回集合的长度或对象中可枚举属性的个数。

参数

1. collection (Array|Object)

   待处理的集合

返回值 (number)

返回集合的大小

示例

```
_.size([1, 2, 3]);
// => 3

_.size({ 'a': 1, 'b': 2 });
// => 2

_.size('pebbles');
// => 7
```

some

- link
- source
- npm

```
_.some(collection, [predicate=_.identity])
```

通过 predicate 检查集合中的元素是否存在任意真值的元素，只要 predicate 返回一次真值，遍历就停止，并返回 true。 predicate 会传入3个参数：(value, index|key, collection)。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (boolean)

返回 true，如果任意元素经 predicate 检查都为真值，否则返回 false。

示例

```
_.some([null, 0, 'yes', false], Boolean);
// => true

var users = [
  { 'user': 'barney', 'active': true },
  { 'user': 'fred',   'active': false }
];

// 使用了 `_.matches` 的回调结果
_.some(users, { 'user': 'barney', 'active': false });
// => false

// 使用了 `_.matchesProperty` 的回调结果
_.some(users, ['active', false]);
// => true

// 使用了 `_.property` 的回调结果
_.some(users, 'active');
// => true
```

sortBy

- link
- source
- npm

```
_.sortBy(collection, [iteratees=[_.identity]])
```

创建一个元素数组。 以 iteratee 处理的结果升序排序。 这个方法执行稳定排序，也就是说相同元素会保持原始排序。 iteratees 会传入1个参数：(value)。

参数

1. collection (Array|Object)

   需要遍历的集合

2. [iteratees=[_.identity]] (...(Function|Function[]|Object|Object[]|string|string[])

   这个函数决定排序

返回值 (Array)

返回排序后的数组

示例

```
var users = [
  { 'user': 'fred',   'age': 48 },
  { 'user': 'barney', 'age': 36 },
  { 'user': 'fred',   'age': 42 },
  { 'user': 'barney', 'age': 34 }
];

_.sortBy(users, function(o) { return o.user; });
// => 排序结果 [['barney', 36], ['barney', 34], ['fred', 48], ['fred

_.sortBy(users, ['user', 'age']);
// => 排序结果 [['barney', 34], ['barney', 36], ['fred', 42], ['fred

_.sortBy(users, 'user', function(o) {
  return Math.floor(o.age / 10);
});
// => 排序结果 [['barney', 36], ['barney', 34], ['fred', 48], ['fred
```

Date

now

- link
- source
- npm

```
_.now()
```

获得 Unix 纪元(1970 1月1日 00:00:00 UTC) 直到现在的毫秒数。

返回值 (number)

返回时间戳

示例

```
_.defer(function(stamp) {
  console.log(_.now() - stamp);
}, _.now());
// => 记录延迟函数调用的毫秒数
```

Function

after

- link
- source
- npm

```
_.after(n, func)
```

反向版 `_.before` 。这个方法创建一个新函数，当调用 `N` 次或者多次之后将触发 `func` 方法。

参数

1. n (number)

   `func` 方法应该在调用多少次后才执行

2. func (Function)

   指定的触发方法

返回值 (Function)

返回限定的函数

示例

```
var saves = ['profile', 'settings'];

var done = _.after(saves.length, function() {
  console.log('done saving!');
});

_.forEach(saves, function(type) {
  asyncSave({ 'type': type, 'complete': done });
});
// => 2次 `asyncSave`之后，输出 'done saving!'。
```

ary

- link
- source
- npm

```
_.ary(func, [n=func.length])
```

创建一个最多接受 `N` 个参数，忽略多余参数的方法。

参数

1. func (Function)

   需要被限制参数个数的函数

2. [n=func.length] (number)

   限制的参数数量

返回值 (Function)

返回新的函数

示例

```
_.map(['6', '8', '10'], _.ary(parseInt, 1));
// => [6, 8, 10]
```

before

- link
- source
- npm

```
_.before(n, func)
```

创建一个调用 `func` 的函数。 调用次数不超过 `N` 次。 之后再调用这个函数，将返回最后一个调用的结果。

参数

1. n (number)

   超过多少次不再调用 `func`

2. func (Function)

   指定的触发的函数

返回值 (Function)

返回限定的函数

示例

```
jQuery(element).on('click', _.before(5, addContactToList));
// => 最多允许添加4个联系人到列表里
```

bind

- link
- source
- npm

```
_.bind(func, thisArg, [partials])
```

创建一个函数 `func` ，这个函数的 `this` 会被绑定在 `thisArg` 。 并且任何附加在 `_.bind` 的参数会被传入到这个绑定函数上。 这个 `_.bind.placeholder` 的值，默认是以 `_` 作为附加部分参数的占位符。

注意: 不同于原生的 Function#bind，这个方法不会设置绑定函数的 length 属性。

参数

1. func (Function)

   要绑定的函数

2. thisArg (*)

   这个 `this` 会被绑定给 `func` 。

3. [partials] (...*)

   附加的部分参数

返回值 (Function)

返回新的绑定函数

示例

```
var greet = function(greeting, punctuation) {
  return greeting + ' ' + this.user + punctuation;
};

var object = { 'user': 'fred' };

var bound = _.bind(greet, object, 'hi');
bound('!');
// => 'hi fred!'

// 使用了占位符
var bound = _.bind(greet, object, _, '!');
bound('hi');
// => 'hi fred!'
```

bindKey

- link
- source
- npm

```
_.bindKey(object, key, [partials])
```

创建一个函数。 该方法绑定 `object[key]` 的方法。 任何附加在 `_.bindKey` 的参数会预设到该绑定函数上。

这个方法与 `_.bind` 的不同之处在于允许重写绑定函数即使它还不存在。 浏览 Peter Michaux's article 了解更多详情。

这个 `_.bindKey.placeholder` 的值，默认是以 `_` 作为附加部分参数的占位符。

参数

1. object (Object)

   需要绑定函数的对象

2. key (string)

   需要绑定函数对象的键

3. [partials] (...*)

   附加的部分参数

返回值 (Function)

返回新的绑定函数

示例

```
var object = {
  'user': 'fred',
  'greet': function(greeting, punctuation) {
    return greeting + ' ' + this.user + punctuation;
  }
};

var bound = _.bindKey(object, 'greet', 'hi');
bound('!');
// => 'hi fred!'

object.greet = function(greeting, punctuation) {
  return greeting + 'ya ' + this.user + punctuation;
};

bound('!');
// => 'hiya fred!'

// 使用了占位符
var bound = _.bindKey(object, 'greet', _, '!');
bound('hi');
// => 'hiya fred!'
```

curry

- link
- source
- npm

```
_.curry(func, [arity=func.length])
```

创建一个函数，该函数接收一个或多个 func 的参数。 当该函数被调用时,如果 func 所需要传递的所有参数都被提供，则直接返回 func 所执行的结果。 否则继续返回该函数并等待接收剩余的参数。 可以使用 func.length 强制需要累积的参数个数。

这个 `_.curry.placeholder` 的值，默认是以 `_` 作为附加部分参数的占位符。

注意: 这个方法不会设置 "length" 到 curried 函数上。

参数

1. func (Function)

   需要 curry 的函数

2. [arity=func.length] (number)

   需要提供给 `func` 的参数数量

返回值 (Function)

返回 curry 后的函数

示例

```
var abc = function(a, b, c) {
  return [a, b, c];
};

var curried = _.curry(abc);

curried(1)(2)(3);
// => [1, 2, 3]

curried(1, 2)(3);
// => [1, 2, 3]

curried(1, 2, 3);
// => [1, 2, 3]

// 使用了占位符
curried(1)(_, 3)(2);
// => [1, 2, 3]
```

curryRight

- link
- source
- npm

```
_.curryRight(func, [arity=func.length])
```

这个方法类似 `_.curry`。 除了它接受参数的方式用 `_.partialRight` 代替了 `_.partial`。

这个 `_.curry.placeholder` 的值，默认是以 `_` 作为附加部分参数的占位符。

注意: 这个方法不会设置 "length" 到 curried 函数上。

参数

1. func (Function)

   需要 curry 的函数

2. [arity=func.length] (number)

   需要提供给 `func` 的参数数量

返回值 (Function)

返回 curry 后的函数

示例

```
var abc = function(a, b, c) {
  return [a, b, c];
};

var curried = _.curryRight(abc);

curried(3)(2)(1);
// => [1, 2, 3]

curried(2, 3)(1);
// => [1, 2, 3]

curried(1, 2, 3);
// => [1, 2, 3]

// 使用了占位符
curried(3)(1, _)(2);
// => [1, 2, 3]
```

debounce

- link
- source
- npm

```
_.debounce(func, [wait=0], [options])
```

创建一个防抖动函数。 该函数会在 `wait` 毫秒后调用 `func` 方法。 该函数提供一个 `cancel` 方法取消延迟的函数调用以及 `flush` 方法立即调用。 可以提供一个 `options` 对象决定如何调用 `func` 方法， options.leading 与|或 options.trailing 决定延迟前后如何触发。 `func` 会传入最后一次传入的参数给防抖动函数。 随后调用的防抖动函数返回是最后一次 `func` 调用的结果。

注意: 如果 `leading` 和 `trailing` 都设定为 true。 则 func 允许 trailing 方式调用的条件为: 在 wait 期间多次调用防抖方法。

查看 David Corbacho's article 了解 `_.debounce` 与 `_.throttle` 的区别。

参数

1. func (Function)

   要防抖动的函数

2. [wait=0] (number)

   需要延迟的毫秒数

3. [options] (Object)

   选项对象

4. [options.leading=false] (boolean)

   指定调用在延迟开始前

5. [options.maxWait] (number)

   设置 `func` 允许被延迟的最大值

6. [options.trailing=true] (boolean)

   指定调用在延迟结束后

返回值 (Function)

返回具有防抖动功能的函数

示例

```
// 避免窗口在变动时出现昂贵的计算开销。
jQuery(window).on('resize', _.debounce(calculateLayout, 150));

// 当点击时 `sendMail` 随后就被调用。
jQuery(element).on('click', _.debounce(sendMail, 300, {
  'leading': true,
  'trailing': false
}));

// 确保 `batchLog` 调用1次之后，1秒内会被触发。
var debounced = _.debounce(batchLog, 250, { 'maxWait': 1000 });
var source = new EventSource('/stream');
jQuery(source).on('message', debounced);

// 取消一个 trailing 的防抖动调用
jQuery(window).on('popstate', debounced.cancel);
```

defer

- link
- source
- npm

```
_.defer(func, [args])
```

延迟调用 `func` 直到当前堆栈清理完毕。 任何附加的参数会传入到 `func`。

参数

1. func (Function)

   要延迟的函数

2. [args] (...*)

   会在调用时传入到 `func` 的参数

返回值 (number)

返回计时器 id

示例

```
_.defer(function(text) {
  console.log(text);
}, 'deferred');
// 一毫秒或更久一些输出 'deferred'。
```

delay

- link
- source
- npm

```
_.delay(func, wait, [args])
```

延迟 `wait` 毫秒后调用 `func`。任何附加的参数会传入到 `func`。

参数

1. func (Function)

   要延迟的函数

2. wait (number)

   要延迟的毫秒数

3. [args] (...*)

   会在调用时传入到 `func` 的参数

返回值 (number)

返回计时器 id

示例

```
_.delay(function(text) {
  console.log(text);
}, 1000, 'later');
// => 一秒后输出 'later'。
```

flip

- link
- source
- npm

```
_.flip(func)
```

创建一个翻转接收参数的 `func` 函数。

参数

1. func (Function)

   要翻转参数的函数

返回值 (Function)

返回新的函数

示例

```
var flipped = _.flip(function() {
  return _.toArray(arguments);
});

flipped('a', 'b', 'c', 'd');
// => ['d', 'c', 'b', 'a']
```

memoize

- link
- source
- npm

```
_.memoize(func, [resolver])
```

创建一个会缓存 `func` 结果的函数。 如果提供了 `resolver` ，就用 `resolver` 的返回值作为 key 缓存函数的结果。 默认情况下用第一个参数作为缓存的 key。 `func` 在调用时 this 会绑定在缓存函数上。

注意: 缓存会暴露在缓存函数的 `cache` 上。 它是可以定制的，只要替换了 _.memoize.Cache 构造函数，或实现了 `Map` 的 `delete` ， `get` ， `has` ， 以及 `set` 方法。

参数

1. func (Function)

   需要缓存化的函数

2. [resolver] (Function)

   这个函数的返回值作为缓存的 key

返回值 (Function)

返回缓存化后的函数

示例

```
var object = { 'a': 1, 'b': 2 };
var other = { 'c': 3, 'd': 4 };

var values = _.memoize(_.values);
values(object);
// => [1, 2]

values(other);
// => [3, 4]

object.a = 2;
values(object);
// => [1, 2]

// 修改结果缓存
values.cache.set(object, ['a', 'b']);
values(object);
// => ['a', 'b']

// 替换 `_.memoize.Cache`
_.memoize.Cache = WeakMap;
```

negate

- link
- source
- npm

```
_.negate(predicate)
```

创建一个对 `func` 结果 取反的函数。 用 predicate 对 `func` 检查的时候， `this` 绑定到创建的函数，并传入对应参数。

参数

1. predicate (Function)

   需要对结果取反的函数

返回值 (Function)

返回一个新函数

示例

```
function isEven(n) {
  return n % 2 == 0;
}

_.filter([1, 2, 3, 4, 5, 6], _.negate(isEven));
// => [1, 3, 5]
```

once

- link
- source
- npm

```
_.once(func)
```

创建一个只能调用一次的函数。 重复调用返回第一次调用的结果。 `func` 调用时，this 绑定到创建的函数，并传入对应参数。

参数

1. func (Function)

   指定的触发的函数

返回值 (Function)

返回受限的函数

示例

```
var initialize = _.once(createApplication);
initialize();
initialize();
// `initialize` 只能调用 `createApplication` 一次。
```

overArgs

- link
- source
- npm

```
_.overArgs(func, [transforms])
```

创建一个函数，调用时 `func` 参数会先一对一的改变。

参数

1. func (Function)

   要包裹的函数

2. [transforms] (...(Function|Function[])

   这个函数会改变传参，单独指定或者指定在数组中

返回值 (Function)

返回新函数

示例

```
function doubled(n) {
  return n * 2;
}

function square(n) {
  return n * n;
}

var func = _.overArgs(function(x, y) {
  return [x, y];
}, square, doubled);

func(9, 3);
// => [81, 6]

func(10, 5);
// => [100, 10]
```

partial

- link
- source
- npm

```
_.partial(func, [partials])
```

创建一个函数。 该函数调用 func，并传入预设的参数。 这个方法类似 `_.bind`，除了它不会绑定 `this`。 这个 `_.partial.placeholder` 的值，默认是以 `_` 作为附加部分参数的占位符。

注意: 这个方法不会设置 "length" 到函数上。

参数

1. func (Function)

   需要预设的函数

2. [partials] (...*)

   预设的参数

返回值 (Function)

返回预设参数的函数

示例

```
var greet = function(greeting, name) {
  return greeting + ' ' + name;
};

var sayHelloTo = _.partial(greet, 'hello');
sayHelloTo('fred');
// => 'hello fred'

// 使用了占位符
var greetFred = _.partial(greet, _, 'fred');
greetFred('hi');
// => 'hi fred'
```

partialRight

- link
- source
- npm

```
_.partialRight(func, [partials])
```

这个函数类似 `_.partial`，除了它是从右到左预设参数的。 这个 *.partialRight.placeholder* 的值，默认是以 作为附加部分参数的占位符。

注意: 这个方法不会设置 "length" 到函数上。

参数

1. func (Function)

   需要预设的函数

2. [partials] (...*)

   预设的参数

返回值 (Function)

返回预设参数的函数

示例

```
var greet = function(greeting, name) {
  return greeting + ' ' + name;
};

var greetFred = _.partialRight(greet, 'fred');
greetFred('hi');
// => 'hi fred'

// 使用了占位符
var sayHelloTo = _.partialRight(greet, 'hello', _);
sayHelloTo('fred');
// => 'hello fred'
```

rearg

- link
- source
- npm

```
_.rearg(func, indexes)
```

创建一个调用 `func` 的函数。 所传递的参数根据 indexes 调整到对应位置。 第一个 index 对应到第一个传参，第二个 index 对应到第二个传参，以此类推。

参数

1. func (Function)

   待调用的函数

2. indexes (...(number|number[])

   重新排列参数的位置，单独指定或者指定在数组中

返回值 (Function)

返回新的函数

示例

```
var rearged = _.rearg(function(a, b, c) {
  return [a, b, c];
}, 2, 0, 1);

rearged('b', 'c', 'a')
// => ['a', 'b', 'c']
```

rest

- link
- source
- npm

```
_.rest(func, [start=func.length-1])
```

创建一个调用 `func` 的函数。 `this` 绑定到这个函数 并且 从 `start` 之后的参数都作为数组传入。

注意: 这个方法基于rest parameter

参数

1. func (Function)

   要应用的函数

2. [start=func.length-1] (number)

   从第几个参数开始应用

返回值 (Function)

返回新的函数

示例

```
var say = _.rest(function(what, names) {
  return what + ' ' + _.initial(names).join(', ') +
    (_.size(names) > 1 ? ', & ' : '') + _.last(names);
});

say('hello', 'fred', 'barney', 'pebbles');
// => 'hello fred, barney, & pebbles'
```

spread

- link
- source
- npm

```
_.spread(func)
```

创建一个调用 `func` 的函数。 `this` 绑定到这个函数上。 把参数作为数组传入，类似于 `Function#apply`

注意: 这个方法基于 spread operator

参数

1. func (Function)

   要应用的函数

返回值 (Function)

返回新的函数

示例

```
var say = _.spread(function(who, what) {
  return who + ' says ' + what;
});

say(['fred', 'hello']);
// => 'fred says hello'

var numbers = Promise.all([
  Promise.resolve(40),
  Promise.resolve(36)
]);

numbers.then(_.spread(function(x, y) {
  return x + y;
}));
// => 返回 76
```

throttle

- link
- source
- npm

```
_.throttle(func, [wait=0], [options])
```

创建一个节流函数，在 `wait` 秒内最多执行 `func` 一次的函数。 该函数提供一个 cancel 方法取消延迟的函数调用以及 flush 方法立即调用。 可以提供一个 options 对象决定如何调用 func 方法， options.leading 与|或 options.trailing 决定 `wait` 前后如何触发。 func 会传入最后一次传入的参数给这个函数。 随后调用的函数返回是最后一次 func 调用的结果。

注意: 如果 leading 和 trailing 都设定为 true。 则 func 允许 trailing 方式调用的条件为: 在 wait 期间多次调用。

查看 David Corbacho's article 了解 `_.throttle` 与 `_.debounce` 的区别

参数

1. func (Function)

   要节流的函数

2. [wait=0] (number)

   需要节流的毫秒

3. [options] (Object)

   选项对象

4. [options.leading=true] (boolean)

   指定调用在节流开始前

5. [options.trailing=true] (boolean)

   指定调用在节流结束后

返回值 (Function)

返回节流的函数

示例

```
// 避免在滚动时过分的更新定位
jQuery(window).on('scroll', _.throttle(updatePosition, 100));

// 点击后就调用 `renewToken`，但5分钟内超过1次。
var throttled = _.throttle(renewToken, 300000, { 'trailing': false
jQuery(element).on('click', throttled);

// 取消一个 trailing 的节流调用
jQuery(window).on('popstate', throttled.cancel);
```

unary

- link
- source
- npm

```
_.unary(func)
```

创建一个最多接受一个参数的函数，忽略多余的参数。

参数

1. func (Function)

   要处理的函数

返回值 (Function)

返回新函数

示例

```
_.map(['6', '8', '10'], _.unary(parseInt));
// => [6, 8, 10]
```

wrap

- link
- source
- npm

```
_.wrap(value, wrapper)
```

创建一个函数。提供的 `value` 包装在 wrapper 函数的第一个参数里。 任何附加的参数都提供给 wrapper 函数。 被调用时 `this` 绑定在创建的函数上。

参数

1. value (*)

   要包装的值

2. wrapper (Function)

   包装函数

返回值 (Function)

返回新的函数

示例

```
var p = _.wrap(_.escape, function(func, text) {
  return '<p>' + func(text) + '</p>';
});

p('fred, barney, & pebbles');
// => '<p>fred, barney, &amp; pebbles</p>'
```

Lang

castArray

- link
- source
- npm

```
_.castArray(value)
```

如果 `value` 不是数组, 那么强制转为数组

参数

1. value (*)

   要处理的值

返回值 (Array)

返回转换后的数组

示例

```
_.castArray(1);
// => [1]

_.castArray({ 'a': 1 });
// => [{ 'a': 1 }]

_.castArray('abc');
// => ['abc']

_.castArray(null);
// => [null]

_.castArray(undefined);
// => [undefined]

_.castArray();
// => []

var array = [1, 2, 3];
console.log(_.castArray(array) === array);
// => true
```

clone

- link
- source
- npm

```
_.clone(value)
```

创建一个 `value` 的浅拷贝。

注意: 这个方法参考自 structured clone algorithm 以及支持 arrays、array buffers、 booleans、 date objects、maps、 numbers， `Object` objects, regexes, sets, strings, symbols, 以及 typed arrays。 参数对象的可枚举属性会拷贝为普通对象。 一些不可拷贝的对象，例如error objects、functions, DOM nodes, 以及 WeakMaps 会返回空对象。

参数

1. value (*)

   要拷贝的值

返回值 (*)

返回拷贝后的值

示例

```
var objects = [{ 'a': 1 }, { 'b': 2 }];

var shallow = _.clone(objects);
console.log(shallow[0] === objects[0]);
// => true
```

cloneDeep

- link
- source
- npm

```
_.cloneDeep(value)
```

这个方法类似 `_.clone`，除了它会递归拷贝 `value`。

参数

1. value (*)

   要深拷贝的值

返回值 (*)

返回拷贝后的值

示例

```
var objects = [{ 'a': 1 }, { 'b': 2 }];

var deep = _.cloneDeep(objects);
console.log(deep[0] === objects[0]);
// => false
```

cloneDeepWith

- link
- source
- npm

```
_.cloneDeepWith(value, [customizer])
```

这个方法类似 `_.cloneWith`，除了它会递归拷贝 `value`。

参数

1. value (*)

   要深拷贝的值

2. [customizer] (Function)

   这个函数定制返回的拷贝值

返回值 (*)

返回拷贝后的值

示例

```
function customizer(value) {
  if (_.isElement(value)) {
    return value.cloneNode(true);
  }
}

var el = _.cloneDeep(document.body, customizer);

console.log(el === document.body);
// => false
console.log(el.nodeName);
// => BODY
console.log(el.childNodes.length);
// => 20
```

cloneWith

* link
* source
* npm

```
_.cloneWith(value, [customizer])
```

这个方法类似 `_.clone`，除了它接受一个 `customizer` 定制返回的拷贝值。 如果 `customizer` 返回 `undefined` 将会拷贝处理方法代替。 `customizer` 会传入5个参数：(value [, index|key, object, stack])

参数

1. value (*)

   要拷贝的值

2. [customizer] (Function)

   这个函数定制返回的拷贝值

返回值 (*)

返回拷贝后的值

示例

```
function customizer(value) {
  if (_.isElement(value)) {
    return value.cloneNode(false);
  }
}

var el = _.cloneWith(document.body, customizer);

console.log(el === document.body);
// => false
console.log(el.nodeName);
// => 'BODY'
console.log(el.childNodes.length);
// => 0
```

eq

- link
- source
- npm

```
_.eq(value, other)
```

执行 `SameValueZero` 比较两者的值确定它们是否相等。

参数

1. value (*)

   要比较的值

2. other (*)

   其他要比较的值

返回值 (boolean)

相等返回 `true`，否则返回 `false`

示例

```
var object = { 'user': 'fred' };
var other = { 'user': 'fred' };

_.eq(object, object);
// => true

_.eq(object, other);
// => false

_.eq('a', 'a');
// => true

_.eq('a', Object('a'));
// => false

_.eq(NaN, NaN);
// => true
```

gt

- link
- source
- npm

```
_.gt(value, other)
```

检查 `value` 是否大于 `other`

参数

1. value (*)

   要比较的值

2. other (*)

   其他要比较的值

返回值 (boolean)

如果 `value` 大于 `other`，返回 `true`，否则返回 `false`

示例

```
_.gt(3, 1);
// => true

_.gt(3, 3);
// => false

_.gt(1, 3);
// => false
```

gte

- link
- source
- npm

```
_.gte(value, other)
```

检查 `value` 是否大于等于 `other`

参数

1. value (*)

   要比较的值

2. other (*)

   其他要比较的值

返回值 (boolean)

如果 `value` 大于等于 `other`，返回 `true`，否则返回 `false`

示例

```
_.gte(3, 1);
// => true

_.gte(3, 3);
// => true

_.gte(1, 3);
// => false
```

isArguments

- link
- source
- npm

```
_.isArguments(value)
```

检查 `value` 是否是 类 `arguments` 对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true` ，否则返回 `false`

示例

```
_.isArguments(function() { return arguments; }());
// => true

_.isArguments([1, 2, 3]);
// => false
```

isArray

- link
- source
- npm

```
_.isArray(value)
```

检查 `value` 是否是 `Array` 类对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isArray([1, 2, 3]);
// => true

_.isArray(document.body.children);
// => false

_.isArray('abc');
// => false

_.isArray(_.noop);
// => false
```

isArrayBuffer

- link
- source
- npm

```
_.isArrayBuffer(value)
```

检查 `value` 是否是 `ArrayBuffer` 对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是 `ArrayBuffer` ，返回 `true` ，否则返回 `false` .

示例

```
_.isArrayBuffer(new ArrayBuffer(2));
// => true

_.isArrayBuffer(new Array(2));
// => false
```

isArrayLike

- link
- source
- npm

```
_.isArrayLike(value)
```

检查 `value` 是否是类数组。 如果是类数组的话，应该不是一个函数，而且 `value.length` 是个整数，大于等于 0，小于或等于 `Number.MAX_SAFE_INTEGER`

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是类数组，返回 `true`，否则返回 `false`

示例

```
_.isArrayLike([1, 2, 3]);
// => true

_.isArrayLike(document.body.children);
// => true

_.isArrayLike('abc');
// => true

_.isArrayLike(_.noop);
// => false
```

isArrayLikeObject

- link
- source
- npm

```
_.isArrayLikeObject(value)
```

这个方法类似 `_.isArrayLike`，除了它还检查值是否是个对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是类数组对象，返回 `true`，否则返回 `false`

示例

```
_.isArrayLikeObject([1, 2, 3]);
// => true

_.isArrayLikeObject(document.body.children);
// => true

_.isArrayLikeObject('abc');
// => false

_.isArrayLikeObject(_.noop);
// => false
```

isBoolean

- link
- source
- npm

```
_.isBoolean(value)
```

检查 `value` 是否是原始 boolean 类型或者对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true` ，否则返回 `false`

示例

```
_.isBoolean(false);
// => true

_.isBoolean(null);
// => false
```

isBuffer

- link
- source
- npm

```
_.isBuffer(value)
```

检查 `value` 是否是个 `buffer`

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isBuffer(new Buffer(2));
// => true

_.isBuffer(new Uint8Array(2));
// => false
```

isDate

- link
- source
- npm

```
_.isDate(value)
```

检查 `value` 是否是 `Date` 类型

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isDate(new Date);
// => true

_.isDate('Mon April 23 2012');
// => false
```

isElement

- link
- source
- npm

```
_.isElement(value)
```

检查 `value` 是否是可能是 DOM 元素

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是 DOM 元素返回 `true`，否则返回 `false`

示例

```
_.isElement(document.body);
// => true

_.isElement('<body>');
// => false
```

isEmpty

- link
- source
- npm

```
_.isEmpty(value)
```

检查 `value` 是否为空。 判断的依据是除非是有枚举属性的对象，length 大于 `0` 的 `arguments` object, array, string 或类jquery选择器。

参数

1. value (Array|Object|string)

   要检查的值

返回值 (boolean)

如果为空返回 `true` ，否则返回 `false`

示例

```
_.isEmpty(null);
// => true

_.isEmpty(true);
// => true

_.isEmpty(1);
// => true

_.isEmpty([1, 2, 3]);
// => false

_.isEmpty({ 'a': 1 });
// => false
```

isEqual

- link
- source
- npm

```
_.isEqual(value, other)
```

执行深比较来决定两者的值是否相等。

注意: 这个方法支持比较 arrays, array buffers, booleans, date objects, error objects, maps, numbers, `Object` objects, regexes, sets, strings, symbols, 以及 typed arrays. `Object` 对象值比较自身的属性，不包括继承的和可枚举的属性。不支持函数和DOM节点。

参数

1. value (*)

   要比较的值

2. other (*)

   其他要比较的值

返回值 (boolean)

如果相等返回 `true` ，否则返回 `false`

示例

```
var object = { 'user': 'fred' };
var other = { 'user': 'fred' };

_.isEqual(object, other);
// => true

object === other;
// => false
```

isEqualWith

- link
- source
- npm

```
_.isEqualWith(value, other, [customizer])
```

这个方法类似 `_.isEqual` 。除了它接受一个 customizer 定制比较值。 如果 customizer 返回 undefined 将会比较处理方法代替。 `customizer` 会传入7个参数：(objValue, othValue [, index|key, object, other, stack])

参数

1. value (*)

   要比较的值

2. other (*)

   其他要比较的值

3. [customizer] (Function)

   这个函数定制比较值

返回值 (boolean)

如果相等返回 `true` ，否则返回 `false`

示例

```
function isGreeting(value) {
  return /^h(?:i|ello)$/.test(value);
}

function customizer(objValue, othValue) {
  if (isGreeting(objValue) && isGreeting(othValue)) {
    return true;
  }
}

var array = ['hello', 'goodbye'];
var other = ['hi', 'goodbye'];

_.isEqualWith(array, other, customizer);
// => true
```

isError

- link
- source
- npm

```
_.isError(value)
```

检查 `value` 是否是 `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, 或 `URIError` object.

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是 error object 返回 `true`，否则返回 `false`

示例

```
_.isError(new Error);
// => true

_.isError(Error);
// => false
```

isFinite

* link
* source
* npm

```
_.isFinite(value)
```

检查 `value` 是否是原始 finite number。

注意: 这个方法基于 `Number.isFinite` .

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是 finite number 返回 `true` ，否则返回 `false`

示例

```
_.isFinite(3);
// => true

_.isFinite(Number.MAX_VALUE);
// => true

_.isFinite(3.14);
// => true

_.isFinite(Infinity);
// => false
```

isFunction

- link
- source
- npm

```
_.isFunction(value)
```

检查 `value` 是否是 `Function` 对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isFunction(_);
// => true

_.isFunction(/abc/);
// => false
```

isInteger

- link
- source
- npm

```
_.isInteger(value)
```

检查 `value` 是否是整数。

注意: 这个方法基于 `Number.isInteger`.

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是整数返回 `true`，否则返回 `false`

示例

```
_.isInteger(3);
// => true

_.isInteger(Number.MIN_VALUE);
// => false

_.isInteger(Infinity);
// => false

_.isInteger('3');
// => false
```

isLength

- link
- source
- npm

```
_.isLength(value)
```

检查 `value` 是否是有效长度

注意: 这个方法参考自 `ToLength` .

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是有效长度返回 `true` ，否则返回 `false`

示例

```
_.isLength(3);
// => true

_.isLength(Number.MIN_VALUE);
// => false

_.isLength(Infinity);
// => false

_.isLength('3');
// => false
```

isMap

- link
- source
- npm

```
_.isMap(value)
```

检查 `value` 是否是个 `Map` 对象

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是 `Map` 对象返回 `true`，否则返回 `false`

示例

```
_.isMap(new Map);
// => true

_.isMap(new WeakMap);
// => false
```

isMatch

- link
- source
- npm

```
_.isMatch(object, source)
```

执行一个深比较来确定 `object` 是否包含有 `source` 的属性值。

注意: 这个方法支持比较相同的值和 `_.isEqual` 一样

参数

1. object (Object)

   要检查的值

2. source (Object)

   匹配包含在 object 的对象

返回值 (boolean)

如果匹配返回 `true` ，否则返回 `false`

示例

```
var object = { 'user': 'fred', 'age': 40 };

_.isMatch(object, { 'age': 40 });
// => true

_.isMatch(object, { 'age': 36 });
// => false
```

isMatchWith

- link
- source
- npm

```
_.isMatchWith(object, source, [customizer])
```

这个方法类似 `_.isMatch` 。 除了它接受一个 customizer 定制比较的值。 如果 customizer 返回 undefined 将会比较处理方法代替。 customizer 会传入5个参数：(objValue, srcValue, index|key, object, source)

参数

1. object (Object)

   要检查的值

2. source (Object)

   匹配包含在 object 的对象

3. [customizer] (Function)

   这个函数定制比较值

返回值 (boolean)

如果匹配返回 `true` ，否则返回 `false`

示例

```
function isGreeting(value) {
  return /^h(?:i|ello)$/.test(value);
}

function customizer(objValue, srcValue) {
  if (isGreeting(objValue) && isGreeting(srcValue)) {
    return true;
  }
}

var object = { 'greeting': 'hello' };
var source = { 'greeting': 'hi' };

_.isMatchWith(object, source, customizer);
// => true
```

isNaN

- link
- source
- npm

```
_.isNaN(value)
```

检查 `value` 是否是 `NaN`.

注意: 这个方法不同于 `isNaN` 对 undefind 和 其他非数值返回 true.

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果符合 `NaN` 返回 `true`，否则返回 `false`

示例

```
_.isNaN(NaN);
// => true

_.isNaN(new Number(NaN));
// => true

isNaN(undefined);
// => true

_.isNaN(undefined);
// => false
```

isNative

- link
- source
- npm

```
_.isNative(value)
```

检查 `value` 是否是原生函数

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是原生函数返回 `true` ，否则返回 `false`

示例

```
_.isNative(Array.prototype.push);
// => true

_.isNative(_);
// => false
```

isNil

- link
- source
- npm

```
_.isNil(value)
```

检查 `value` 是否是 `null` 或者 `undefined` 。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是 `null` 或者 `undefined` 返回 `true` ，否则返回 `false`

示例

```
_.isNil(null);
// => true

_.isNil(void 0);
// => true

_.isNil(NaN);
// => false
```

isNull

- link
- source
- npm

```
_.isNull(value)
```

检查 `value` 是否是 `null` .

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是 `null` 返回 `true` ，否则返回 `false`

示例

```
_.isNull(null);
// => true

_.isNull(void 0);
// => false
```

isNumber

- link
- source
- npm

```
_.isNumber(value)
```

检查 `value` 是否是原始数值型 或者 对象。

注意: 要排除 `Infinity` , `-Infinity` , 以及 `NaN` 数值类型，用 `_.isFinite` 方法

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true` ，否则返回 `false`

示例

```
_.isNumber(3);
// => true

_.isNumber(Number.MIN_VALUE);
// => true

_.isNumber(Infinity);
// => true

_.isNumber('3');
// => false
```

isObject

- link
- source
- npm

```
_.isObject(value)
```

检查 `value` 是否是 `Object` 的 language type。 (例如： arrays, functions, objects, regexes, `new Number(0)` , 以及 `new String('')`)

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是对象返回 `true` ，否则返回 `false`

示例

```
_.isObject({});
// => true

_.isObject([1, 2, 3]);
// => true

_.isObject(_.noop);
// => true

_.isObject(null);
// => false
```

isObjectLike

- link
- source
- npm

```
_.isObjectLike(value)
```

检查 `value` 是否是 类对象。 类对象应该不是 `null` 以及 `typeof` 的结果是 "object"。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是类对象返回 `true`，否则返回 `false`

示例

```
_.isObjectLike({});
// => true

_.isObjectLike([1, 2, 3]);
// => true

_.isObjectLike(_.noop);
// => false

_.isObjectLike(null);
// => false
```

isPlainObject

- link
- source
- npm

```
_.isPlainObject(value)
```

检查 `value` 是否是普通对象。 也就是说该对象由 `Object` 构造函数创建或者 `[[Prototype]]` 为空。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是普通对象返回 `true`，否则返回 `false`

示例

```
function Foo() {
  this.a = 1;
}

_.isPlainObject(new Foo);
// => false

_.isPlainObject([1, 2, 3]);
// => false

_.isPlainObject({ 'x': 0, 'y': 0 });
// => true

_.isPlainObject(Object.create(null));
// => true
```

isRegExp

- link
- source
- npm

```
_.isRegExp(value)
```

检查 `value` 是否是 `RegExp` 对象

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isRegExp(/abc/);
// => true

_.isRegExp('/abc/');
// => false
```

isSafeInteger

- link
- source
- npm

```
_.isSafeInteger(value)
```

检查 `value` 是否是安全整数。 这个整数应该是符合 IEEE-754 标准的非双精度浮点数。

注意: 这个方法基于 `Number.isSafeInteger`.

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是安全整数返回 `true`，否则返回 `false`

示例

```
_.isSafeInteger(3);
// => true

_.isSafeInteger(Number.MIN_VALUE);
// => false

_.isSafeInteger(Infinity);
// => false

_.isSafeInteger('3');
// => false
```

isSet

- link
- source
- npm

```
_.isSet(value)
```

检查 `value` 是否是 `Set` 对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isSet(new Set);
// => true

_.isSet(new WeakSet);
// => false
```

isString

- link
- source
- npm

```
_.isString(value)
```

检查 `value` 是否是原始字符串或者对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isString('abc');
// => true

_.isString(1);
// => false
```

isSymbol

- link
- source
- npm

```
_.isSymbol(value)
```

检查 `value` 是否是原始 `Symbol` 或者对象。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isSymbol(Symbol.iterator);
// => true

_.isSymbol('abc');
// => false
```

isTypedArray

- link
- source
- npm

```
_.isTypedArray(value)
```

检查 `value` 是否是TypedArray。

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isTypedArray(new Uint8Array);
// => true

_.isTypedArray([]);
// => false
```

isUndefined

- link
- source
- npm

```
_.isUndefined(value)
```

检查 `value` 是否是 `undefined`.

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isUndefined(void 0);
// => true

_.isUndefined(null);
// => false
```

isWeakMap

- link
- source
- npm

```
_.isWeakMap(value)
```

检查 `value` 是否是 `WeakMap` 对象

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true` ，否则返回 `false`

示例

```
_.isWeakMap(new WeakMap);
// => true

_.isWeakMap(new Map);
// => false
```

isWeakSet

- link
- source
- npm

```
_.isWeakSet(value)
```

检查 `value` 是否是 `WeakSet` 对象

参数

1. value (*)

   要检查的值

返回值 (boolean)

如果是正确的类型，返回 `true`，否则返回 `false`

示例

```
_.isWeakSet(new WeakSet);
// => true

_.isWeakSet(new Set);
// => false
```

lt

* link
* source
* npm

```
_.lt(value, other)
```

检查 `value` 是否是 小于 `other` 。

参数

1. value (*)

   要比较的值

2. other (*)

   其他要比较的值

返回值 (boolean)

如果 `value` 小于 `other` 返回 `true`，否则返回 `false`

示例

```
_.lt(1, 3);
// => true

_.lt(3, 3);
// => false

_.lt(3, 1);
// => false
```

lte

- link
- source
- npm

```
_.lte(value, other)
```

检查 `value` 是否是 小于等于 `other` .

参数

1. value (*)

   要比较的值

2. other (*)

   其他要比较的值

返回值 (boolean)

如果 `value` 小于等于 `other` 返回 `true` ，否则返回 `false`

示例

```
_.lte(1, 3);
// => true

_.lte(3, 3);
// => true

_.lte(3, 1);
// => false
```

toArray

- link
- source
- npm

```
_.toArray(value)
```

转换 `value` 为数组

参数

1. value (*)

   要转换的值

返回值 (Array)

然后转换后的数组

示例

```
_.toArray({ 'a': 1, 'b': 2 });
// => [1, 2]

_.toArray('abc');
// => ['a', 'b', 'c']

_.toArray(1);
// => []

_.toArray(null);
// => []
```

toInteger

- link
- source
- npm

```
_.toInteger(value)
```

转换 `value` 为整数

注意: 这个函数参考 `ToInteger`.

参数

1. value (*)

   要转换的值

返回值 (number)

返回转换后的整数

示例

```
_.toInteger(3);
// => 3

_.toInteger(Number.MIN_VALUE);
// => 0

_.toInteger(Infinity);
// => 1.7976931348623157e+308

_.toInteger('3');
// => 3
```

toLength

- link
- source
- npm

```
_.toLength(value)
```

转换 `value` 为用作类数组对象的长度整数。

注意: 这个方法基于 `ToLength`.

参数

1. value (*)

   要转换的值

返回值 (number)

返回转换后的整数

示例

```
_.toLength(3);
// => 3

_.toLength(Number.MIN_VALUE);
// => 0

_.toLength(Infinity);
// => 4294967295

_.toLength('3');
// => 3
```

toNumber

- link
- source
- npm

```
_.toNumber(value)
```

转换 `value` 为数值

参数

1. value (*)

   要处理的值

返回值 (number)

返回数值

示例

```
_.toNumber(3);
// => 3

_.toNumber(Number.MIN_VALUE);
// => 5e-324

_.toNumber(Infinity);
// => Infinity

_.toNumber('3');
// => 3
```

toPlainObject

- link
- source
- npm

```
_.toPlainObject(value)
```

转换 `value` 为普通对象。 包括继承的可枚举属性。

参数

1. value (*)

   要转换的值

返回值 (Object)

返回转换后的普通对象

示例

```
function Foo() {
  this.b = 2;
}

Foo.prototype.c = 3;

_.assign({ 'a': 1 }, new Foo);
// => { 'a': 1, 'b': 2 }

_.assign({ 'a': 1 }, _.toPlainObject(new Foo));
// => { 'a': 1, 'b': 2, 'c': 3 }
```

toSafeInteger

- link
- source
- npm

```
_.toSafeInteger(value)
```

转换 `value` 为安全整数。 安全整数可以用于比较和准确的表示。

参数

1. value (*)

   要转换的值

返回值 (number)

返回转换后的整数

示例

```
_.toSafeInteger(3);
// => 3

_.toSafeInteger(Number.MIN_VALUE);
// => 0

_.toSafeInteger(Infinity);
// => 9007199254740991

_.toSafeInteger('3');
// => 3
```

toString

- link
- source
- npm

```
_.toString(value)
```

如果 `value` 不是字符串，将其转换为字符串。 `null` 和 `undefined` 将返回空字符串。

参数

1. value (*)

   要转换的值

返回值 (string)

返回字符串

示例

```
_.toString(null);
// => ''

_.toString(-0);
// => '-0'

_.toString([1, 2, 3]);
// => '1,2,3'
```

Math

add

- link
- source
- npm

```
_.add(augend, addend)
```

相加两个数

参数

1. augend (number)

   相加的第一个数

2. addend (number)

   相加的第二个数

返回值 (number)

返回总和

示例

```
_.add(6, 4);
// => 10
```

ceil

- link
- source
- npm

```
_.ceil(number, [precision=0])
```

根据 `precision` 向上舍入 `number` 。

参数

1. number (number)

   要向上舍入的值

2. [precision=0] (number)

   精度

返回值 (number)

返回向上舍入的结果

示例

```
_.ceil(4.006);
// => 5

_.ceil(6.004, 2);
// => 6.01

_.ceil(6040, -2);
// => 6100
```

floor

- link
- source
- npm

```
_.floor(number, [precision=0])
```

根据 `precision` 向下保留 `number` 。

参数

1. number (number)

   要向下保留的数

2. [precision=0] (number)

   精度

返回值 (number)

返回向下保留的结果

示例

```
_.floor(4.006);
// => 4

_.floor(0.046, 2);
// => 0.04

_.floor(4060, -2);
// => 4000
```

max

- link
- source
- npm

```
_.max(array)
```

计算 `array` 中最大的值。 如果 `array` 是 空的或者假值将会返回 undefined。

参数

1. array (Array)

   要计算的数组

返回值 (*)

返回最大的值

示例

```
_.max([4, 2, 8, 6]);
// => 8

_.max([]);
// => undefined
```

maxBy

- link
- source
- npm

```
_.maxBy(array, [iteratee=_.identity])
```

这个方法类似 `_.max` 除了它接受 `iteratee` 调用每一个元素，根据返回的 value 决定排序准则。 iteratee 会传入1个参数：(value)。

参数

1. array (Array)

   要遍历的数组

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (*)

返回最大值

示例

```
var objects = [{ 'n': 1 }, { 'n': 2 }];

_.maxBy(objects, function(o) { return o.n; });
// => { 'n': 2 }

// 使用了 `_.property` iteratee 的回调结果
_.maxBy(objects, 'n');
// => { 'n': 2 }
```

mean

- link
- source
- npm

```
_.mean(array)
```

计算 `array` 的平均值。

参数

1. array (Array)

   要遍历的数组

返回值 (number)

返回平均值

示例

```
_.mean([4, 2, 8, 6]);
// => 5
```

min

- link
- source
- npm

```
_.min(array)
```

计算 array 中最小的值。 如果 array 是 空的或者假值将会返回 undefined。

参数

1. array (Array)

   要计算的数组

返回值 (*)

返回最小值

示例

```
_.min([4, 2, 8, 6]);
// => 2

_.min([]);
// => undefined
```

minBy

- link
- source
- npm

```
_.minBy(array, [iteratee=_.identity])
```

这个方法类似 `_.min` 。 除了它接受 iteratee 调用每一个元素，根据返回的 value 决定排序准则。 iteratee 会传入1个参数：(value)。

参数

1. array (Array)

   要遍历的数组

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (*)

返回最小值

示例

```
var objects = [{ 'n': 1 }, { 'n': 2 }];

_.minBy(objects, function(o) { return o.n; });
// => { 'n': 1 }

// 使用了 `_.property` iteratee 的回调结果
_.minBy(objects, 'n');
// => { 'n': 1 }
```

round

- link
- source
- npm

```
_.round(number, [precision=0])
```

根据 precision 四舍五入 number。

参数

1. number (number)

   要四舍五入的值

2. [precision=0] (number)

   精度

返回值 (number)

返回四舍五入的结果

示例

```
_.round(4.006);
// => 4

_.round(4.006, 2);
// => 4.01

_.round(4060, -2);
// => 4100
```

subtract

- link
- source
- npm

```
_.subtract(minuend, subtrahend)
```

两双相减

参数

1. minuend (number)

   相减的第一个数

2. subtrahend (number)

   相减的第二个数

返回值 (number)

返回结果

示例

```
_.subtract(6, 4);
// => 2
```

sum

- link
- source
- npm

```
_.sum(array)
```

计算 `array` 中值的总和

参数

1. array (Array)

   要遍历的数组

返回值 (number)

返回总和

示例

```
_.sum([4, 2, 8, 6]);
// => 20
```

sumBy

- link
- source
- npm

```
_.sumBy(array, [iteratee=_.identity])
```

这个方法类似 `_.sum` 。除了它接受 iteratee 调用每一个元素，根据返回的 value 决定如何计算。 iteratee 会传入1个参数：(value)。

参数

1. array (Array)

   要遍历的数组

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (number)

返回总和

示例

```
var objects = [{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }];

_.sumBy(objects, function(o) { return o.n; });
// => 20

// 使用了 `_.property` 的回调结果
_.sumBy(objects, 'n');
// => 20
```

Methods

templateSettings.imports._

- link
- source

```
_.templateSettings.imports._
```

`lodash` 函数的引用

Number

clamp

- link
- source
- npm

```
_.clamp(number, [min], max)
```

返回限制在 `min` 和 `max` 之间的值

参数

1. number (number)

   被限制的值

2. [min] (number)

   最小绝对值

3. max (number)

   最大绝对值

返回值 (number)

[min, max] 中的一个

示例

```
_.clamp(-10, -5, 5);
// => -5

_.clamp(10, -5, 5);
// => 5
```

inRange

- link
- source
- npm

```
_.inRange(number, [start=0], end)
```

检查 `n` 是否在 `start` 与 `end` 之间，但不包括 `end` 。 如果 `end` 没有指定，那么 `start` 设置为0。 如果 `start` 大于 `end` ，那么参数会交换以便支持负范围。

参数

1. number (number)

   要检查的值

2. [start=0] (number)

   开始范围

3. end (number)

   结束范围

返回值 (boolean)

如果值在范围内返回 `true` ，否则返回 `false`

示例

```
_.inRange(3, 2, 4);
// => true

_.inRange(4, 8);
// => true

_.inRange(4, 2);
// => false

_.inRange(2, 2);
// => false

_.inRange(1.2, 2);
// => true

_.inRange(5.2, 4);
// => false

_.inRange(-3, -2, -6);
// => true
```

random

- link
- source
- npm

```
_.random([min=0], [max=1], [floating])
```

产生一个包括 `min` 与 `max` 之间的数。 如果只提供一个参数返回一个0到提供数之间的数。 如果 `floating` 设为 true，或者 `min` 或 `max` 是浮点数，结果返回浮点数。

注意: JavaScript 遵循 IEEE-754 标准处理无法预料的浮点数结果。

参数

1. [min=0] (number)

   最小值

2. [max=1] (number)

   最大值

3. [floating] (boolean)

   是否返回浮点数

返回值 (number)

返回随机数

示例

```
_.random(0, 5);
// =>  0 和 5 之间的数

_.random(5);
// => 同样是 0 和 5 之间的数

_.random(5, true);
// => 0 和 5 之间的浮点数

_.random(1.2, 5.2);
// =>  1.2 和 5.2 之间的浮点数
```

Object

assign

- link
- source
- npm

```
_.assign(object, [sources])
_.assign(object, [sources])
```

分配来源对象的可枚举属性到目标对象上。 来源对象的应用规则是从左到右，随后的下一个对象的属性会覆盖上一个对象的属性。

注意: 这方法会改变源对象，参考自 `Object.assign`.

参数

1. object (Object)

   目标对象

2. [sources] (...Object)

   来源对象

返回值 (Object)

返回对象

示例

```
function Foo() {
  this.c = 3;
}

function Bar() {
  this.e = 5;
}

Foo.prototype.d = 4;
Bar.prototype.f = 6;

_.assign({ 'a': 1 }, new Foo, new Bar);
// => { 'a': 1, 'c': 3, 'e': 5 }
```

assignIn extend

- link
- source
- npm

```
_.assignIn(object, [sources])
```

这个方法类似 `_.assign`。除了它会遍历并继承来源对象的属性。

注意: 这方法会改变源对象

参数

1. object (Object)

   目标对象

2. [sources] (...Object)

   来源对象

返回值 (Object)

返回对象

示例

```
function Foo() {
  this.b = 2;
}

function Bar() {
  this.d = 4;
}

Foo.prototype.c = 3;
Bar.prototype.e = 5;

_.assignIn({ 'a': 1 }, new Foo, new Bar);
// => { 'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5 }
```

assignInWith extendWith

- link
- source
- npm

```
_.assignInWith(object, sources, [customizer])
```

这个方法类似 `_.assignIn`。 除了它接受一个 customizer `决定如何分配值。 如果` customizer `返回` undefined `将会由分配处理方法代` customizer` 会传入5个参数：(objValue, srcValue, key, object, source)。

注意: 这方法会改变源对象

参数

1. object (Object)

   目标对象

2. sources (...Object)

   来源对象

3. [customizer] (Function)

   这个函数决定分配的值

返回值 (Object)

返回对象

示例

```
function customizer(objValue, srcValue) {
  return _.isUndefined(objValue) ? srcValue : objValue;
}

var defaults = _.partialRight(_.assignInWith, customizer);

defaults({ 'a': 1 }, { 'b': 2 }, { 'a': 3 });
// => { 'a': 1, 'b': 2 }
```

assignWith

- link
- source
- npm

```
_.assignWith(object, sources, [customizer])
```

这个方法类似 `_.assign` 。 除了它接受一个 customizer `决定如何分配值。 如果` customizer `返回` undefined `将会由分配处理方法代` customizer` 会传入5个参数：(objValue, srcValue, key, object, source)。

注意: 这方法会改变源对象

参数

1. object (Object)

   目标对象

2. sources (...Object)

   来源对象

3. [customizer] (Function)

   这个函数决定分配的值

返回值 (Object)

返回对象

示例

```
function customizer(objValue, srcValue) {
  return _.isUndefined(objValue) ? srcValue : objValue;
}

var defaults = _.partialRight(_.assignWith, customizer);

defaults({ 'a': 1 }, { 'b': 2 }, { 'a': 3 });
// => { 'a': 1, 'b': 2 }
```

at

- link
- source
- npm

```
_.at(object, [paths])
```

根据 `object` 的路径获取值为数组。

参数

1. object (Object)

   要遍历的对象

2. [paths] (...(string|string[])

   要获取的对象的元素路径，单独指定或者指定在数组中

返回值 (Array)

返回选中值的数组

示例

```
var object = { 'a': [{ 'b': { 'c': 3 } }, 4] };

_.at(object, ['a[0].b.c', 'a[1]']);
// => [3, 4]

_.at(['a', 'b', 'c'], 0, 2);
// => ['a', 'c']
```

create

- link
- source
- npm

```
_.create(prototype, [properties])
```

创建一个继承 `prototype` 的对象。 如果提供了 `properties` ，它的可枚举属性会被分配到创建的对象上。

参数

1. prototype (Object)

   要继承的对象

2. [properties] (Object)

   待分配的属性

返回值 (Object)

返回新对象

示例

```
function Shape() {
  this.x = 0;
  this.y = 0;
}

function Circle() {
  Shape.call(this);
}

Circle.prototype = _.create(Shape.prototype, {
  'constructor': Circle
});

var circle = new Circle;
circle instanceof Circle;
// => true

circle instanceof Shape;
// => true
```

defaults

- link
- source
- npm

```
_.defaults(object, [sources])
```

分配来源对象的可枚举属性到目标对象所有解析为 `undefined` 的属性上。 来源对象从左到右应用。 一旦设置了相同属性的值，后续的将被忽略掉。

注意: 这方法会改变源对象

参数

1. object (Object)

   目标对象

2. [sources] (...Object)

   来源对象

返回值 (Object)

返回对象

示例

```
_.defaults({ 'user': 'barney' }, { 'age': 36 }, { 'user': 'fred' })
// => { 'user': 'barney', 'age': 36 }
```

defaultsDeep

- link
- source
- npm

```
_.defaultsDeep(object, [sources])
```

这个方法类似 `_.defaults` ，除了它会递归分配默认属性。

注意: 这方法会改变源对象

参数

1. object (Object)

   目标对象

2. [sources] (...Object)

   来源对象

返回值 (Object)

返回对象

示例

```
_.defaultsDeep({ 'user': { 'name': 'barney' } }, { 'user': { 'name
// => { 'user': { 'name': 'barney', 'age': 36 } }
```

findKey

- link
- source
- npm

```
_.findKey(object, [predicate=_.identity])
```

这个方法类似 `_.find` 。除了它返回最先被 `predicate` 判断为真值的元素 key，而不是元素本身。

参数

1. object (Object)

   需要检索的对象

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (string|undefined)

返回匹配的 key，否则返回 `undefined` 。

示例

```
var users = {
  'barney':  { 'age': 36, 'active': true },
  'fred':    { 'age': 40, 'active': false },
  'pebbles': { 'age': 1,  'active': true }
};

_.findKey(users, function(o) { return o.age < 40; });
// => 'barney' (无法保证遍历的顺序)

// 使用了 `_.matches` 的回调结果
_.findKey(users, { 'age': 1, 'active': true });
// => 'pebbles'

// 使用了 `_.matchesProperty` 的回调结果
_.findKey(users, ['active', false]);
// => 'fred'

// 使用了 `_.property` 的回调结果
_.findKey(users, 'active');
// => 'barney'
```

findLastKey

- link
- source
- npm

```
_.findLastKey(object, [predicate=_.identity])
```

这个方法类似 `_.findKey` 。不过它是反方向开始遍历的。

参数

1. object (Object)

   需要检索的对象

2. [predicate=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (string|undefined)

返回匹配的 key，否则返回 `undefined` 。

示例

```
var users = {
  'barney':  { 'age': 36, 'active': true },
  'fred':    { 'age': 40, 'active': false },
  'pebbles': { 'age': 1,  'active': true }
};

_.findLastKey(users, function(o) { return o.age < 40; });
// => 返回 'pebbles'， `_.findKey` 会返回 'barney'

// 使用了 `_.matches` 的回调结果
_.findLastKey(users, { 'age': 36, 'active': true });
// => 'barney'

// 使用了 `_.matchesProperty` 的回调结果
_.findLastKey(users, ['active', false]);
// => 'fred'

// 使用了 `_.property` 的回调结果
_.findLastKey(users, 'active');
// => 'pebbles'
```

forIn

- link
- source
- npm

```
_.forIn(object, [iteratee=_.identity])
```

使用 `iteratee` 遍历对象的自身和继承的可枚举属性。 iteratee 会传入3个参数：(value, key, object)。 如果返回 false，iteratee 会提前退出遍历。

参数

1. object (Object)

   要遍历的对象

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

返回值 (Object)

返回对象

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forIn(new Foo, function(value, key) {
  console.log(key);
});
// => 输出 'a', 'b', 然后 'c' (无法保证遍历的顺序)
```

forInRight

- link
- source
- npm

```
_.forInRight(object, [iteratee=_.identity])
```

这个方法类似 `_.forIn` 。除了它是反方向开始遍历的。

参数

1. object (Object)

   要遍历的对象

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

返回值 (Object)

返回对象

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forInRight(new Foo, function(value, key) {
  console.log(key);
});
// => 输出 'c', 'b', 然后 'a'， `_.forIn` 会输出 'a', 'b', 然后 'c'
```

forOwn

- link
- source
- npm

```
_.forOwn(object, [iteratee=_.identity])
```

使用 `iteratee` 遍历自身的可枚举属性。 iteratee 会传入3个参数：(value, key, object)。 如果返回 false，iteratee 会提前退出遍历。

参数

1. object (Object)

   要遍历的对象

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

返回值 (Object)

返回对象

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forOwn(new Foo, function(value, key) {
  console.log(key);
});
// => 输出 'a' 然后 'b' (无法保证遍历的顺序)
```

forOwnRight

- link
- source
- npm

```
_.forOwnRight(object, [iteratee=_.identity])
```

这个方法类似 `_.forOwn` 。除了它是反方向开始遍历的。

参数

1. object (Object)

   要遍历的对象

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

返回值 (Object)

返回对象

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forOwnRight(new Foo, function(value, key) {
  console.log(key);
});
// => 输出 'b' 然后 'a'， `_.forOwn` 会输出 'a' 然后 'b'
```

functions

- link
- source
- npm

```
_.functions(object)
```

返回一个 function 对象自身可枚举属性名的数组。

参数

1. object (Object)

   要检索的对象

返回值 (Array)

返回包含属性名的新数组

示例

```
function Foo() {
  this.a = _.constant('a');
  this.b = _.constant('b');
}

Foo.prototype.c = _.constant('c');

_.functions(new Foo);
// => ['a', 'b']
```

functionsIn

- link
- source
- npm

```
_.functionsIn(object)
```

返回一个 function 对象自身和继承的可枚举属性名的数组。

参数

1. object (Object)

   要检索的对象

返回值 (Array)

返回包含属性名的新数组

示例

```
function Foo() {
  this.a = _.constant('a');
  this.b = _.constant('b');
}

Foo.prototype.c = _.constant('c');

_.functionsIn(new Foo);
// => ['a', 'b', 'c']
```

get

- link
- source
- npm

```
_.get(object, path, [defaultValue])
```

根据对象路径获取值。 如果解析 value 是 `undefined` 会以 `defaultValue` 取代。

参数

1. object (Object)

   要检索的对象

2. path (Array|string)

   要获取的对象路径

3. [defaultValue] (*)

   如果解析值是 `undefined` ，这值会被返回

返回值 (*)

返回解析的值

示例

```
var object = { 'a': [{ 'b': { 'c': 3 } }] };

_.get(object, 'a[0].b.c');
// => 3

_.get(object, ['a', '0', 'b', 'c']);
// => 3

_.get(object, 'a.b.c', 'default');
// => 'default'
```

has

- link
- source
- npm

```
_.has(object, path)
```

检查 `path` 是否是对象的直接属性。

参数

1. object (Object)

   要检索的对象

2. path (Array|string)

   要检查的路径

返回值 (boolean)

如果存在返回 true，否则返回 `false`

示例

```
var object = { 'a': { 'b': { 'c': 3 } } };
var other = _.create({ 'a': _.create({ 'b': _.create({ 'c': 3 }) })

_.has(object, 'a');
// => true

_.has(object, 'a.b.c');
// => true

_.has(object, ['a', 'b', 'c']);
// => true

_.has(other, 'a');
// => false
```

hasIn

- link
- source
- npm

```
_.hasIn(object, path)
```

检查 `path` 是否是对象的直接 或者 继承属性。

参数

1. object (Object)

   要检索的对象

2. path (Array|string)

   要检查的路径

返回值 (boolean)

如果存在返回 true，否则返回 `false`

示例

```
var object = _.create({ 'a': _.create({ 'b': _.create({ 'c': 3 }) ]

_.hasIn(object, 'a');
// => true

_.hasIn(object, 'a.b.c');
// => true

_.hasIn(object, ['a', 'b', 'c']);
// => true

_.hasIn(object, 'b');
// => false
```

invert

- link
- source
- npm

```
_.invert(object, [multiVal])
```

创建一个键值倒置的对象。 如果 `object` 有重复的值，后面的值会覆盖前面的值。 如果 `multiVal` 为 true，重复的值则组成数组。

参数

1. object (Object)

   要倒置的对象

2. [multiVal] (boolean)

   每个 key 允许多个值

返回值 (Object)

返回新的倒置的对象

示例

```
var object = { 'a': 1, 'b': 2, 'c': 1 };

_.invert(object);
// => { '1': 'c', '2': 'b' }
```

invertBy

- link
- source
- npm

```
_.invertBy(object, [iteratee=_.identity])
```

这个方法类似 `_.invert` 。除了它接受 iteratee 调用每一个元素，可在返回值中定制key。 iteratee 会传入1个参数：(value)。

参数

1. object (Object)

   要倒置的对象

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会调用每一个元素

返回值 (Object)

返回新的倒置的对象

示例

```
var object = { 'a': 1, 'b': 2, 'c': 1 };

_.invertBy(object);
// => { '1': ['a', 'c'], '2': ['b'] }

_.invertBy(object, function(value) {
  return 'group' + value;
});
// => { 'group1': ['a', 'c'], 'group2': ['b'] }
```

invoke

- link
- source
- npm

```
_.invoke(object, path, [args])
```

调用对象路径的方法

参数

1. object (Object)

   要检索的对象

2. path (Array|string)

   要调用方法的路径

3. [args] (...*)

   调用方法的参数

返回值 (*)

返回调用方法的结果

var object = { 'a': [{ 'b': { 'c': [1, 2, 3, 4] } }] };

_.invoke(object, 'a[0].b.c.slice', 1, 3); // => [2, 3]

keys

- link
- source
- npm

```
_.keys(object)
```

创建 `object` 自身可枚举属性名为一个数组。

注意: 非对象的值会被强制转换为对象，查看 ES spec 了解详情

参数

1. object (Object)

   要检索的对象

返回值 (Array)

返回包含属性名的数组

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.keys(new Foo);
// => ['a', 'b'] (无法保证遍历的顺序)

_.keys('hi');
// => ['0', '1']
```

keysIn

- link
- source
- npm

```
_.keysIn(object)
```

创建 `object` 自身 或 继承的可枚举属性名为一个数组。

注意: 非对象的值会被强制转换为对象

参数

1. object (Object)

   要检索的对象

返回值 (Array)

返回包含属性名的数组

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.keysIn(new Foo);
// => ['a', 'b', 'c'] (无法保证遍历的顺序)
```

mapKeys

- link
- source
- npm

```
_.mapKeys(object, [iteratee=_.identity])
```

反向版 `_.mapValues`。 这个方法创建一个对象，对象的值与源对象相同，但 key 是通过 `iteratee` 产生的。

参数

1. object (Object)

   要遍历的对象

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Object)

返回映射后的新对象

示例

```
_.mapKeys({ 'a': 1, 'b': 2 }, function(value, key) {
  return key + value;
});
// => { 'a1': 1, 'b2': 2 }
```

mapValues

- link
- source
- npm

```
_.mapValues(object, [iteratee=_.identity])
```

创建一个对象，对象的key相同，值是通过 `iteratee` 产生的。 iteratee 会传入3个参数： (value, key, object)

参数

1. object (Object)

   要遍历的对象

2. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Object)

返回映射后的对象

示例

```
var users = {
  'fred':    { 'user': 'fred',    'age': 40 },
  'pebbles': { 'user': 'pebbles', 'age': 1 }
};

_.mapValues(users, function(o) { return o.age; });
// => { 'fred': 40, 'pebbles': 1 } (无法保证遍历的顺序)

// 使用了 `_.property` 的回调结果
_.mapValues(users, 'age');
// => { 'fred': 40, 'pebbles': 1 } (无法保证遍历的顺序)
```

merge

- link
- source
- npm

```
_.merge(object, [sources])
```

递归合并来源对象的自身和继承的可枚举属性到目标对象。 跳过来源对象解析为 `undefined` 的属性。 数组和普通对象会递归合并，其他对象和值会被直接分配。 来源对象从左到右分配，后续的来源对象属性会覆盖之前分配的属性。

注意: 这方法会改变源对象

参数

1. object (Object)

   目标对象

2. [sources] (...Object)

   来源对象

返回值 (Object)

返回对象

示例

```
var users = {
  'data': [{ 'user': 'barney' }, { 'user': 'fred' }]
};

var ages = {
  'data': [{ 'age': 36 }, { 'age': 40 }]
};

_.merge(users, ages);
// => { 'data': [{ 'user': 'barney', 'age': 36 }, { 'user': 'fred',
```

mergeWith

- link
- source
- npm

```
_.mergeWith(object, sources, customizer)
```

这个方法类似 `_.merge` 。除了它接受一个 `customizer` 决定如何合并。 如果 `customizer` 返回 `undefined` 将会由合并处理方法代替。

参数

1. object (Object)

   目标对象

2. sources (...Object)

   来源对象

3. customizer (Function)

   这个方法决定如何合并

返回值 (Object)

返回对象

示例

```
function customizer(objValue, srcValue) {
  if (_.isArray(objValue)) {
    return objValue.concat(srcValue);
  }
}

var object = {
  'fruits': ['apple'],
  'vegetables': ['beet']
};

var other = {
  'fruits': ['banana'],
  'vegetables': ['carrot']
};

_.mergeWith(object, other, customizer);
// => { 'fruits': ['apple', 'banana'], 'vegetables': ['beet', 'carr
```

omit

- link
- source
- npm

```
_.omit(object, [props])
```

反向版 `_.pick` 。 这个方法返回忽略属性之外的自身和继承的可枚举属性。

参数

1. object (Object)

   来源对象

2. [props] (...(string|string[])

   要被忽略的属性，单独指定或指定在数组中

返回值 (Object)

返回新对象

示例

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.omit(object, ['a', 'c']);
// => { 'b': '2' }
```

omitBy

- link
- source
- npm

```
_.omitBy(object, [predicate=_.identity])
```

反向版 `_.pickBy`。 这个方法返回经 `predicate` 判断不是真值的属性的自身和继承的可枚举属性。

参数

1. object (Object)

   来源对象

2. [predicate=_.identity] (Function|Object|string)

   这个函数会调用每一个属性

返回值 (Object)

返回新对象

示例

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.omitBy(object, _.isNumber);
// => { 'b': '2' }
```

pick

- link
- source
- npm

```
_.pick(object, [props])
```

创建一个从 `object` 中选中的属性的对象。

参数

1. object (Object)

   来源对象

2. [props] (...(string|string[])

   要选中的属性名，单独指定或指定在数组中

返回值 (Object)

返回新对象

示例

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.pick(object, ['a', 'c']);
// => { 'a': 1, 'c': 3 }
```

pickBy

- link
- source
- npm

```
_.pickBy(object, [predicate=_.identity])
```

创建一个从 `object` 中经 `predicate` 判断为真值的属性为对象。 predicate 会传入1个参数：(value)

参数

1. object (Object)

   来源对象

2. [predicate=_.identity] (Function|Object|string)

   这个函数会调用每一个属性

返回值 (Object)

返回新对象

示例

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.pickBy(object, _.isNumber);
// => { 'a': 1, 'c': 3 }
```

result

- link
- source
- npm

```
_.result(object, path, [defaultValue])
```

这个方法类似 `_.get` 。 除了如果解析到的值是一个函数的话，就绑定 `this` 到这个函数并返回执行后的结果。

参数

1. object (Object)

   要检索的对象

2. path (Array|string)

   要解析的属性路径

3. [defaultValue] (*)

   如果值是 `undefined` ，返回这个值

返回值 (*)

返回解析后的值

示例

```
var object = { 'a': [{ 'b': { 'c1': 3, 'c2': _.constant(4) } }] };

_.result(object, 'a[0].b.c1');
// => 3

_.result(object, 'a[0].b.c2');
// => 4

_.result(object, 'a[0].b.c3', 'default');
// => 'default'

_.result(object, 'a[0].b.c3', _.constant('default'));
// => 'default'
```

set

- link
- source
- npm

```
_.set(object, path, value)
```

设置值到对象对应的属性路径上，如果没有则创建这部分路径。 缺少的索引属性会创建为数组，而缺少的属性会创建为对象。 使用 `_.setWith` 定制创建。

参数

1. object (Object)

   要修改的对象

2. path (Array|string)

   要设置的对象路径

3. value (*)

   要设置的值

返回值 (Object)

返回对象

示例

```
var object = { 'a': [{ 'b': { 'c': 3 } }] };

_.set(object, 'a[0].b.c', 4);
console.log(object.a[0].b.c);
// => 4

_.set(object, 'x[0].y.z', 5);
console.log(object.x[0].y.z);
// => 5
```

setWith

- link
- source
- npm

```
_.setWith(object, path, value, [customizer])
```

这个方法类似 `_.set` 。除了它接受一个 `customizer` 决定如何设置对象路径的值。如果 `customizer` 返回 `undefined` 将会有它的处理方法代替。 `customizer` 会传入3个参数：(nsValue, key, nsObject) 注意: 这个方法会改变源对象

参数

1. object (Object)

   要修改的对象

2. path (Array|string)

   要设置的对象路径

3. value (*)

   要设置的值

4. [customizer] (Function)

   这个函数决定如何分配值

返回值 (Object)

返回对象

示例

```
_.setWith({ '0': { 'length': 2 } }, '[0][1][2]', 3, Object);
// => { '0': { '1': { '2': 3 }, 'length': 2 } }
```

toPairs

- link
- source
- npm

```
_.toPairs(object)
```

创建一个对象自身可枚举属性的键值对数组。

参数

1. object (Object)

   要检索的对象

返回值 (Array)

返回键值对的数组

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.toPairs(new Foo);
// => [['a', 1], ['b', 2]] (无法保证遍历的顺序)
```

toPairsIn

- link
- source
- npm

```
_.toPairsIn(object)
```

创建一个对象自身和继承的可枚举属性的键值对数组。

参数

1. object (Object)

   要检索的对象

返回值 (Array)

返回键值对的数组

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.toPairsIn(new Foo);
// => [['a', 1], ['b', 2], ['c', 1]] (无法保证遍历的顺序)
```

transform

- link
- source
- npm

```
_.transform(object, [iteratee=_.identity], [accumulator])
```

`_.reduce` 的代替方法。 这个方法会改变对象为一个新的 `accumulator` 对象，来自每一次经 `iteratee` 处理自身可枚举对象的结果。 每次调用可能会改变 `accumulator` 对象。 iteratee 会传入4个对象：(accumulator, value, key, object)。 如果返回 `false` ，iteratee 会提前退出。

参数

1. object (Array|Object)

   要遍历的对象

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

3. [accumulator] (*)

   定制叠加的值

返回值 (*)

返回叠加后的值

示例

```
_.transform([2, 3, 4], function(result, n) {
  result.push(n *= n);
  return n % 2 == 0;
}, []);
// => [4, 9]

_.transform({ 'a': 1, 'b': 2, 'c': 1 }, function(result, value, key
  (result[value] || (result[value] = [])).push(key);
}, {});
// => { '1': ['a', 'c'], '2': ['b'] }
```

unset

- link
- source
- npm

```
_.unset(object, path)
```

移除对象路径的属性。 注意: 这个方法会改变源对象

参数

1. object (Object)

   要修改的对象

2. path (Array|string)

   要移除的对象路径

返回值 (boolean)

移除成功返回 `true`，否则返回 `false`

示例

```
var object = { 'a': [{ 'b': { 'c': 7 } }] };
_.unset(object, 'a[0].b.c');
// => true

console.log(object);
// => { 'a': [{ 'b': {} }] };

_.unset(object, 'a[0].b.c');
// => true

console.log(object);
// => { 'a': [{ 'b': {} }] };
```

values

- link
- source
- npm

```
_.values(object)
```

创建 `object` 自身可枚举属性的值为数组

注意: 非对象的值会强制转换为对象

参数

1. object (Object)

   要检索的对象

返回值 (Array)

返回对象属性的值的数组

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.values(new Foo);
// => [1, 2] (无法保证遍历的顺序)

_.values('hi');
// => ['h', 'i']
```

valuesIn

- link
- source
- npm

```
_.valuesIn(object)
```

创建 `object` 自身和继承的可枚举属性的值为数组

注意: 非对象的值会强制转换为对象

参数

1. object (Object)

   要检索的对象

返回值 (Array)

Returns the array of property values.

示例

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.valuesIn(new Foo);
// => [1, 2, 3] (无法保证遍历的顺序)
```

Properties

templateSettings

- link
- source
- npm

```
_.templateSettings
```

(Object): 默认情况下，这些都是用于处理lodash的模板，类似 Ruby 的嵌入式 (ERB)。 可以改变接下来的设置用新的方式代替。

templateSettings.escape

- link
- source

```
_.templateSettings.escape
```

(RegExp): 用于检测要进行HTML转义 `data` 的属性值。

templateSettings.evaluate

- link
- source

```
_.templateSettings.evaluate
```

(RegExp): 用于检测表达式代码

templateSettings.imports

- link
- source

```
_.templateSettings.imports
```

(Object): 用于导入变量到编译模板

templateSettings.interpolate

- link
- source

```
_.templateSettings.interpolate
```

(RegExp): 用于检测要插入的 `data` 的属性值。

templateSettings.variable

- link
- source

```
_.templateSettings.variable
```

(string): 用于引用模板文本中的 `data` 对象

VERSION

- link
- source

```
_.VERSION
```

(string): 语义化版本号

Seq

_

- link
- source
- npm

```
_(value)
```

创建一个经 `lodash` 包装后的对象会启用隐式链。返回的数组、集合、方法相互之间能够链式调用。 检索唯一值或返回原始值会自动解除链条并返回计算后的值，否则需要调用 `_#value` 方法解除链(即获得计算结果)。

显式链式调用，在任何情况下需要先用 `_#value` 解除链后，才能使用 `_.chain` 开启。

链式方法是惰性计算的，直到隐式或者显式调用了 `_#value` 才会执行计算。

惰性计算接受几种支持 shortcut fusion 的方法， shortcut fusion 是一种通过合并链式 iteratee 调用从而大大降低迭代的次数以提高执行性能的方式。 部分链有资格 shortcut fusion，如果它至少有超过二百个元素的数组和任何只接受一个参数的 iteratees。 触发的方式是任何一个 shortcut fusion 有了变化。

链式方法支持定制版本，只要 `_#value` 包含或者间接包含在版本中。

除了 lodash 的自身方法，包装后的对象还支持 `Array` 的 `String` 的方法。

支持 `Array` 的方法: `concat` , `join` , `pop` , `push` , `shift` , `sort` , `splice` , 以及 `unshift`

支持 `String` 的方法: `replace` 以及 `split`

支持 shortcut fusion 的方法: `at` , `compact` , `drop` , `dropRight` , `dropWhile` , `filter` , `find` , `findLast` , `head` , `initial` , `last` , `map` , `reject` , `reverse` , `slice` , `tail` , `take` , `takeRight` , `takeRightWhile` , `takeWhile` , 以及 `toArray`

默认不支持 链式调用 的方法: `add` , `attempt` , `camelCase` , `capitalize` , `ceil` , `clamp` , `clone` , `cloneDeep` , `cloneDeepWith` , `cloneWith` , `deburr` , `endsWith` , `eq` , `escape` , `escapeRegExp` , `every` , `find` , `findIndex` , `findKey` , `findLast` , `findLastIndex` , `findLastKey` , `floor` , `forEach` , `forEachRight` , `forIn` , `forInRight` , `forOwn` , `forOwnRight` , `get` , `gt` , `gte` , `has` , `hasIn` , `head` , `identity` , `includes` , `indexOf` , `inRange` , `invoke` , `isArguments` , `isArray` , `isArrayBuffer` , `isArrayLike` , `isArrayLikeObject` , `isBoolean` , `isBuffer` , `isDate` , `isElement` , `isEmpty` , `isEqual` , `isEqualWith` , `isError` , `isFinite` , `isFunction` , `isInteger` , `isLength` , `isMap` ,

`isMatch` , `isMatchWith` , `isNaN` , `isNative` , `isNil` , `isNull` , `isNumber` , `isObject` , `isObjectLike` , `isPlainObject` , `isRegExp` , `isSafeInteger` , `isSet` , `isString` , `isUndefined` , `isTypedArray` , `isWeakMap` , `isWeakSet` , `join` , `kebabCase` , `last` , `lastIndexOf` , `lowerCase` , `lowerFirst` , `lt` , `lte` , `max` , `maxBy` , `mean` , `min` , `minBy` , `noConflict` , `noop` , `now` , `pad` , `padEnd` , `padStart` , `parseInt` , `pop` , `random` , `reduce` , `reduceRight` , `repeat` , `result` , `round` , `runInContext` , `sample` , `shift` , `size` , `snakeCase` , `some` , `sortedIndex` , `sortedIndexBy` , `sortedLastIndex` , `sortedLastIndexBy` , `startCase` , `startsWith` , `subtract` , `sum` , `sumBy` , `template` , `times` , `toLower` , `toInteger` , `toLength` , `toNumber` , `toSafeInteger` , `toString` , `toUpper` , `trim` , `trimEnd` , `trimStart` , `truncate` , `unescape` , `uniqueId` , `upperCase` , `upperFirst` , `value` , 以及 `words`

支持 链式调用 的方法: `after` , `ary` , `assign` , `assignIn` , `assignInWith` , `assignWith` , `at` , `before` , `bind` , `bindAll` , `bindKey` , `castArray` , `chain` , `chunk` , `commit` , `compact` , `concat` , `conforms` , `constant` , `countBy` , `create` , `curry` , `debounce` , `defaults` , `defaultsDeep` , `defer` , `delay` , `difference` , `differenceBy` , `differenceWith` , `drop` , `dropRight` , `dropRightWhile` , `dropWhile` , `fill` , `filter` , `flatten` , `flattenDeep` , `flattenDepth` , `flip` , `flow` , `flowRight` , `fromPairs` , `functions` , `functionsIn` , `groupBy` , `initial` , `intersection` , `intersectionBy` , `intersectionWith` , `invert` , `invertBy` , `invokeMap` , `iteratee` , `keyBy` , `keys` , `keysIn` , `map` , `mapKeys` , `mapValues` , `matches` , `matchesProperty` , `memoize` , `merge` , `mergeWith` , `method` , `methodOf` , `mixin` , `negate` , `nthArg` , `omit` , `omitBy` , `once` , `orderBy` , `over` , `overArgs` , `overEvery` , `overSome` , `partial` , `partialRight` , `partition` , `pick` , `pickBy` , `plant` , `property` , `propertyOf` , `pull` , `pullAll` , `pullAllBy` , `pullAt` , `push` , `range` , `rangeRight` , `rearg` , `reject` , `remove` , `rest` , `reverse` , `sampleSize` , `set` , `setWith` , `shuffle` , `slice` , `sort` , `sortBy` , `splice` , `spread` , `tail` , `take` , `takeRight` , `takeRightWhile` , `takeWhile` , `tap` , `throttle` , `thru` , `toArray` , `toPairs` , `toPairsIn` , `toPath` , `toPlainObject` , `transform` , `unary` , `union` , `unionBy` , `unionWith` , `uniq` , `uniqBy` , `uniqWith` , `unset` , `unshift` , `unzip` , `unzipWith` , `values` , `valuesIn` , `without` , `wrap` , `xor` , `xorBy` , `xorWith` , `zip` , `zipObject` , `zipObjectDeep` , 以及 `zipWith`

参数

1. value (*)

   需要被包装为 `lodash` 实例的值.

返回值 (Object)

返回 `lodash` 包装后的实例

示例

```
function square(n) {
  return n * n;
}

var wrapped = _([1, 2, 3]);

// 返回未包装的值
wrapped.reduce(_.add);
// => 6

// 返回链式包装的值
var squares = wrapped.map(square);

_.isArray(squares);
// => false

_.isArray(squares.value());
// => true
```

chain

- link
- source
- npm

```
_.chain(value)
```

创建一个经 `lodash` 包装的对象以启用显式链模式，要解除链必须使用 `_#value` 方法。

参数

1. value (*)

    要包装的值

返回值 (Object)

返回 `lodash` 包装的实例

示例

```
var users = [
  { 'user': 'barney',  'age': 36 },
  { 'user': 'fred',    'age': 40 },
  { 'user': 'pebbles', 'age': 1 }
];

var youngest = _
  .chain(users)
  .sortBy('age')
  .map(function(o) {
    return o.user + ' is ' + o.age;
  })
  .head()
  .value();
// => 'pebbles is 1'
```

prototype.at

- link
- source
- npm

```
_.prototype.at([paths])
```

这个方法是 `_.at` 的包装版本

参数

1. [paths] (...(string|string[])

   要选择元素的属性路径， 单独指定或者数组

返回值 (Object)

返回 `lodash` 的包装实例

示例

```
var object = { 'a': [{ 'b': { 'c': 3 } }, 4] };

_(object).at(['a[0].b.c', 'a[1]']).value();
// => [3, 4]

_(['a', 'b', 'c']).at(0, 2).value();
// => ['a', 'c']
```

prototype.chain

- link
- source
- npm

```
_.prototype.chain()
```

开启包装对象的显式链。

返回值 (Object)

返回 `lodash` 的包装实例

示例

```
var users = [
  { 'user': 'barney', 'age': 36 },
  { 'user': 'fred',   'age': 40 }
];

// 不启用显式链
_(users).head();
// => { 'user': 'barney', 'age': 36 }

// 启用显式链
_(users)
  .chain()
  .head()
  .pick('user')
  .value();
// => { 'user': 'barney' }
```

prototype.commit

- link
- source
- npm

```
_.prototype.commit()
```

执行链式队列并返回结果

返回值 (Object)

返回 `lodash` 的包装实例

示例

```
var array = [1, 2];
var wrapped = _(array).push(3);

console.log(array);
// => [1, 2]

wrapped = wrapped.commit();
console.log(array);
// => [1, 2, 3]

wrapped.last();
// => 3

console.log(array);
// => [1, 2, 3]
```

prototype.next

- link
- source
- npm

```
_.prototype.next()
```

获得包装对象的下一个值，遵循 iterator 协议。

返回值 (Object)

返回下一个 iterator 值

示例

```
var wrapped = _([1, 2]);

wrapped.next();
// => { 'done': false, 'value': 1 }

wrapped.next();
// => { 'done': false, 'value': 2 }

wrapped.next();
// => { 'done': true, 'value': undefined }
```

prototype.plant

- link
- source
- npm

```
_.prototype.plant(value)
```

创建一个链式队列的拷贝，传入的值作为链式队列的值。

参数

1. value (*)

   替换原值的值

返回值 (Object)

返回 `lodash` 的包装实例

示例

```
function square(n) {
  return n * n;
}

var wrapped = _([1, 2]).map(square);
var other = wrapped.plant([3, 4]);

other.value();
// => [9, 16]

wrapped.value();
// => [1, 4]
```

prototype.Symbol.iterator

- link
- source
- npm

```
_.prototype.Symbol.iterator()
```

启用包装对象为 iterable。

返回值 (Object)

返回包装对象

示例

```
var wrapped = _([1, 2]);

wrapped[Symbol.iterator]() === wrapped;
// => true

Array.from(wrapped);
// => [1, 2]
```

prototype.value run, toJSON, valueOf

- link
- source
- npm

```
_.prototype.value()
```

执行链式队列并提取解链后的值

返回值 (*)

返回解链后的值

示例

```
_([1, 2, 3]).value();
// => [1, 2, 3]
```

tap

- link
- source
- npm

```
_.tap(value, interceptor)
```

这个方法调用一个 `interceptor` 并返回 `value`。`interceptor` 传入一个参数：(value) 目的是 `进入` 链的中间以便执行操作。

参数

1. value (*)

   提供给 `interceptor` 的值

2. interceptor (Function)

   调用函数

返回值 (*)

返回 `value`

示例

```
_([1, 2, 3])
 .tap(function(array) {
   // 改变传入的数组
   array.pop();
 })
 .reverse()
 .value();
// => [2, 1]
```

thru

- link
- source
- npm

```
_.thru(value, interceptor)
```

这个方法类似 `_.tap`，除了它返回 `interceptor` 的返回结果

参数

1. value (*)

   提供给 `interceptor` 的值

2. interceptor (Function)

   调用函数

返回值 (*)

返回 `interceptor` 的返回结果

示例

```
_('  abc  ')
 .chain()
 .trim()
 .thru(function(value) {
   return [value];
 })
 .value();
// => ['abc']
```

wrapperFlatMap

- link
- source
- npm

```
_.wrapperFlatMap([iteratee=_.identity])
```

这个方法是 `_.flatMap` 的包装版本。

参数

1. [iteratee=_.identity] (Function|Object|string)

   这个函数会处理每一个元素

返回值 (Object)

返回 `lodash` 的包装实例

示例

```
function duplicate(n) {
  return [n, n];
}

_([1, 2]).flatMap(duplicate).value();
// => [1, 1, 2, 2]
```

String

camelCase

- link
- source
- npm

```
_.camelCase([string=''])
```

转换字符串为 驼峰写法

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回驼峰写法的字符串

示例

```
_.camelCase('Foo Bar');
// => 'fooBar'

_.camelCase('--foo-bar');
// => 'fooBar'

_.camelCase('__foo_bar__');
// => 'fooBar'
```

capitalize

- link
- source
- npm

```
_.capitalize([string=''])
```

转换字符串首字母为大写，剩下为小写。

参数

1. [string=''] (string)

   要大写开头的字符串

返回值 (string)

返回大写开头的字符串

示例

```
_.capitalize('FRED');
// => 'Fred'
```

deburr

- link
- source
- npm

```
_.deburr([string=''])
```

转换 latin-1 supplementary letters#Character_table) 为基本拉丁字母，并删除变音符。

参数

1. [string=''] (string)

   要处理的字符串

返回值 (string)

返回处理后的字符串

示例

```
_.deburr('déjà vu');
// => 'deja vu'
```

endsWith

- link
- source
- npm

```
_.endsWith([string=''], [target], [position=string.length])
```

检查给定的字符是否是字符串的结尾

参数

1. [string=''] (string)

   要检索的字符串

2. [target] (string)

   要检索字符

3. [position=string.length] (number)

   检索的位置

返回值 (boolean)

如果是结尾返回 `true`，否则返回 `false`

示例

```
_.endsWith('abc', 'c');
// => true

_.endsWith('abc', 'b');
// => false

_.endsWith('abc', 'b', 2);
// => true
```

escape

- link
- source
- npm

```
_.escape([string=''])
```

转义字符 "&", "<", ">", """, "'", 以及 "`" 为HTML实体字符。

注意: 不会转义其他字符，如果需要，可以使用第三方库，例如 *he*。

虽然 ">" 是对称转义的，像是 ">" 和 "/" 没有特殊的意义，所以不需要在 HTML 中转义。 除非它们是标签的一部分，或者是不带引号的属性值。 查看 Mathias Bynens 的文章 (under "semi-related fun fact") 了解详情

在 IE < 9 中转义引号，因为会中断属性值或 HTML 注释，查看 HTML5 安全列表的 #59, #102, #108, 以及 #133 了解详情

当解析为 HTML 时应该总是 引用属性值 以减少 XSS 的可能性。

参数

1. [string=''] (string)

   要转义的字符串

返回值 (string)

返回转义后的字符串

示例

```
_.escape('fred, barney, & pebbles');
// => 'fred, barney, &amp; pebbles'
```

escapeRegExp

- link
- source
- npm

```
_.escapeRegExp([string=''])
```

转义 `RegExp` 中特殊的字符 "^", "$", "\", ".", "*", "+", "?", "(", ")", "[", "]", "{", "}", 以及 "|"。

参数

1. [string=''] (string)

   要转义的字符串

返回值 (string)

返回转义后的字符串

示例

```
_.escapeRegExp('[lodash](https://lodash.com/)');
// => '\[lodash\]\(https://lodash\.com/\)'
```

kebabCase

- link
- source
- npm

```
_.kebabCase([string=''])
```

转换字符串为 kebab case。

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回转换后的字符串

示例

```
_.kebabCase('Foo Bar');
// => 'foo-bar'

_.kebabCase('fooBar');
// => 'foo-bar'

_.kebabCase('__foo_bar__');
// => 'foo-bar'
```

lowerCase

- link
- source
- npm

```
_.lowerCase([string=''])
```

以空格分开单词并转换为小写。

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回小写的字符串

示例

```
_.lowerCase('--Foo-Bar');
// => 'foo bar'

_.lowerCase('fooBar');
// => 'foo bar'

_.lowerCase('__FOO_BAR__');
// => 'foo bar'
```

lowerFirst

- link
- source
- npm

```
_.lowerFirst([string=''])
```

转换首字母为小写。

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回转换后的字符串

示例

```
_.lowerFirst('Fred');
// => 'fred'

_.lowerFirst('FRED');
// => 'fRED'
```

pad

- link
- source
- npm

```
_.pad([string=''], [length=0], [chars=' '])
```

如果字符串长度小于 `length` 则从左到右填充字符。 如果没法平均分配，则截断超出的长度。

参数

1. [string=''] (string)

   要填充的字符串

2. [length=0] (number)

   填充的长度

3. [chars=' '] (string)

   填充字符

返回值 (string)

返回填充后的字符串

示例

```
_.pad('abc', 8);
// => '  abc   '

_.pad('abc', 8, '_-');
// => '_-abc_-_'

_.pad('abc', 3);
// => 'abc'
```

padEnd

- link
- source
- npm

```
_.padEnd([string=''], [length=0], [chars=' '])
```

如果字符串长度小于 length 则在右侧填充字符。 如果超出长度则截断超出的部分。

参数

1. [string=''] (string)

   要填充的字符串

2. [length=0] (number)

   填充的长度

3. [chars=' '] (string)

   填充字符

返回值 (string)

Returns 返回填充后的字符串

示例

```
_.padEnd('abc', 6);
// => 'abc   '

_.padEnd('abc', 6, '_-');
// => 'abc_-_'

_.padEnd('abc', 3);
// => 'abc'
```

padStart

- link
- source
- npm

```
_.padStart([string=''], [length=0], [chars=' '])
```

如果字符串长度小于 length 则在左侧填充字符。 如果超出长度则截断超出的部分。

参数

1. [string=''] (string)

   要填充的字符串

2. [length=0] (number)

   填充的长度

3. [chars=' '] (string)

   填充字符

返回值 (string)

Returns 返回填充后的字符串

示例

```
_.padStart('abc', 6);
// => '   abc'

_.padStart('abc', 6, '_-');
// => '_-_abc'

_.padStart('abc', 3);
// => 'abc'
```

parseInt

- link
- source
- npm

```
_.parseInt(string, [radix])
```

以指定的基数转换字符串为整数。 如果基数是 `undefined` 或者 0，则基数默认是10，如果字符串是16进制，则基数为16。

注意: 这个方法与 ES5 implementation 的 `parseInt` 一致

参数

1. string (string)

   要转换的字符串

2. [radix] (number)

   基数

返回值 (number)

返回转换后的整数

示例

```
_.parseInt('08');
// => 8

_.map(['6', '08', '10'], _.parseInt);
// => [6, 8, 10]
```

repeat

- link
- source
- npm

```
_.repeat([string=''], [n=0])
```

重复 N 次字符串

参数

1. [string=''] (string)

   要重复的字符串

2. [n=0] (number)

   重复的次数

返回值 (string)

返回重复的字符串

示例

```
_.repeat('*', 3);
// => '***'

_.repeat('abc', 2);
// => 'abcabc'

_.repeat('abc', 0);
// => ''
```

replace

- link
- source
- npm

```
_.replace([string=''], pattern, 要替换的内容)
```

替换字符串中匹配的内容为给定的内容

注意: 这个方法基于 `String#replace`

参数

1. [string=''] (string)

   待替换的字符串

2. pattern (RegExp|string)

   要匹配的内容

3. 要替换的内容 (Function|string)

返回值 (string)

返回替换完成的字符串

示例

```
_.replace('Hi Fred', 'Fred', 'Barney');
// => 'Hi Barney'
```

snakeCase

- link
- source
- npm

```
_.snakeCase([string=''])
```

转换字符串为 snake case

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回转换后的字符串

示例

```
_.snakeCase('Foo Bar');
// => 'foo_bar'

_.snakeCase('fooBar');
// => 'foo_bar'

_.snakeCase('--foo-bar');
// => 'foo_bar'
```

split

- link
- source
- npm

```
_.split([string=''], [separator], [limit])
```

以 `separator` 拆分字符串

注意: 这个方法基于 `String#split`

参数

1. [string=''] (string)

   要拆分的字符串
2. [separator] (RegExp|string)

   拆分的分隔符
3. [limit] (number)

   限制的数量

返回值 (Array)

返回拆分部分的字符串的数组

示例

```
_.split('a-b-c', '-', 2);
// => ['a', 'b']
```

startCase

- link
- source
- npm

```
_.startCase([string=''])
```

转换字符串为 start case

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回转换后的字符串

示例

```
_.startCase('--foo-bar');
// => 'Foo Bar'

_.startCase('fooBar');
// => 'Foo Bar'

_.startCase('__foo_bar__');
// => 'Foo Bar'
```

startsWith

- link
- source
- npm

```
_.startsWith([string=''], [target], [position=0])
```

检查字符串是否以 `target` 开头。

参数

1. [string=''] (string)

   要检索的字符串

2. [target] (string)

   要检查的字符串

3. [position=0] (number)

   检索的位置

返回值 (boolean)

如果符合条件返回 `true`，否则返回 `false`

示例

```
_.startsWith('abc', 'a');
// => true

_.startsWith('abc', 'b');
// => false

_.startsWith('abc', 'b', 1);
// => true
```

template

- link
- source
- npm

```
_.template([string=''], [options])
```

创建一个预编译模板方法，可以插入数据到模板中 "interpolate" 分隔符相应的位置。 HTML会在 "escape" 分隔符中转换为相应实体。 在 "evaluate" 分隔符中允许执行JavaScript代码。 在模板中可以自由访问变量。 如果设置了选项对象，则会优先覆盖 `_.templateSettings` 的值。

注意: 在开发过程中可以使用 sourceURLs 便于调试。

了解更多预编译模板的信息查看 lodash的自定义构建文档

了解更多 Chrome 沙箱扩展的信息查看 Chrome的扩展文档

参数

1. [string=''] (string)

   模板字符串

2. [options] (Object)

   选项对象

3. [options.escape] (RegExp)

   "escape" 分隔符

4. [options.evaluate] (RegExp)

   "evaluate" 分隔符

5. [options.imports] (Object)

   导入对象到模板中作为自由变量

6. [options.interpolate] (RegExp)

   "interpolate" 分隔符

7. [options.sourceURL] (string)

   模板编译的来源URL

8. [options.variable] (string)

数据对象的变量名

返回值 (Function)

返回编译模板函数

示例

```
// 使用 "interpolate" 分隔符创建编译模板
var compiled = _.template('hello <%= user %>!');
compiled({ 'user': 'fred' });
// => 'hello fred!'

// 使用 HTML "escape" 转义数据的值
var compiled = _.template('<b><%- value %></b>');
compiled({ 'value': '<script>' });
// => '<b>&lt;script&gt;</b>'

// 使用 "evaluate" 分隔符执行 JavaScript 和 生成HTML代码
var compiled = _.template('<% _.forEach(users, function(user) { %><
compiled({ 'users': ['fred', 'barney'] });
// => '<li>fred</li><li>barney</li>'

// 在 "evaluate" 分隔符中使用内部的 `print` 函数
var compiled = _.template('<% print("hello " + user); %>!');
compiled({ 'user': 'barney' });
// => 'hello barney!'

// 使用 ES 分隔符代替默认的 "interpolate" 分隔符
var compiled = _.template('hello ${ user }!');
compiled({ 'user': 'pebbles' });
// => 'hello pebbles!'

// 使用自定义的模板分隔符
_.templateSettings.interpolate = /{{([\s\S]+?)}}/g;
var compiled = _.template('hello {{ user }}!');
compiled({ 'user': 'mustache' });
// => 'hello mustache!'

// 使用反斜杠符号作为纯文本处理
var compiled = _.template('<%= "\\<%- value %\\>" %>');
compiled({ 'value': 'ignored' });
// => '<%- value %>'

// 使用 `imports` 选项导入 `jq` 作为 `jQuery` 的别名
var text = '<% jq.each(users, function(user) { %><li><%- user %></l
var compiled = _.template(text, { 'imports': { 'jq': jQuery } });
compiled({ 'users': ['fred', 'barney'] });
// => '<li>fred</li><li>barney</li>'
```

```
// 使用 `sourceURL` 选项指定模板的来源URL
var compiled = _.template('hello <%= user %>!', { 'sourceURL': '/ba
compiled(data);
// => 在开发工具的 Sources 选项卡 或 Resources 面板中找到 "greeting.jst'

// 使用 `variable` 选项确保在编译模板中不声明变量
var compiled = _.template('hi <%= data.user %>!', { 'variable': 'da
compiled.source;
// => function(data) {
//   var __t, __p = '';
//   __p += 'hi ' + ((__t = ( data.user )) == null ? '' : __t) + '
//   return __p;
// }

// 使用 `source` 特性内联编译模板
// 便以查看行号、错误信息、堆栈
fs.writeFileSync(path.join(cwd, 'jst.js'), '\
  var JST = {\
    "main": ' + _.template(mainText).source + '\
  };\
');
```

toLower

- link
- source
- npm

```
_.toLower([string=''])
```

转换整体的字符串为小写

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回小写的字符串

示例

```
_.toLower('--Foo-Bar');
// => '--foo-bar'

_.toLower('fooBar');
// => 'foobar'

_.toLower('__FOO_BAR__');
// => '__foo_bar__'
```

toUpper

- link
- source
- npm

```
_.toUpper([string=''])
```

转换整体的字符串为大写

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回大写的字符串

示例

```
_.toUpper('--foo-bar');
// => '--FOO-BAR'

_.toUpper('fooBar');
// => 'FOOBAR'

_.toUpper('__foo_bar__');
// => '__FOO_BAR__'
```

trim

- link
- source
- npm

```
_.trim([string=''], [chars=whitespace])
```

从字符串中移除前面和后面的空白 或 指定的字符。

参数

1. [string=''] (string)

   要处理的字符串

2. [chars=whitespace] (string)

   要处理的字符

返回值 (string)

返回处理后的字符串

示例

```
_.trim('  abc  ');
// => 'abc'

_.trim('-_-abc-_-', '_-');
// => 'abc'

_.map(['  foo  ', '  bar  '], _.trim);
// => ['foo', 'bar']
```

trimEnd

- link
- source
- npm

```
_.trimEnd([string=''], [chars=whitespace])
```

移除字符串后面的空白 或 指定的字符。

参数

1. [string=''] (string)

   要处理的字符串

2. [chars=whitespace] (string)

   要处理的字符

返回值 (string)

返回处理后的字符串

示例

```
_.trimEnd('  abc  ');
// => '  abc'

_.trimEnd('-_-abc-_-', '_-');
// => '-_-abc'
```

trimStart

- link
- source
- npm

```
_.trimStart([string=''], [chars=whitespace])
```

移除字符串中前面的空白 或 指定的字符。

参数

1. [string=''] (string)

   要处理的字符串

2. [chars=whitespace] (string)

   要处理的字符

返回值 (string)

返回处理后的字符串

示例

```
_.trimStart('  abc  ');
// => 'abc  '

_.trimStart('-_-abc-_-', '_-');
// => 'abc-_-'
```

truncate

- link
- source
- npm

```
_.truncate([string=''], [options])
```

截断字符串，如果字符串超出了限定的最大值。 被截断的字符串后面会以 `omission` 代替， `omission` 默认是 "..."。

参数

1. [string=''] (string)

   要截断的字符串

2. [options] (Object)

   选项对象

3. [options.length=30] (number)

   允许的最大长度

4. [options.omission='...'] (string)

   超出后的代替字符

5. [options.separator] (RegExp|string)

   截断点

返回值 (string)

返回截断后的字符串

示例

```
_.truncate('hi-diddly-ho there, neighborino');
// => 'hi-diddly-ho there, neighbo...'

_.truncate('hi-diddly-ho there, neighborino', {
  'length': 24,
  'separator': ' '
});
// => 'hi-diddly-ho there,...'

_.truncate('hi-diddly-ho there, neighborino', {
  'length': 24,
  'separator': /,? +/
});
// => 'hi-diddly-ho there...'

_.truncate('hi-diddly-ho there, neighborino', {
  'omission': ' [...]'
});
// => 'hi-diddly-ho there, neig [...]'
```

unescape

- link
- source
- npm

```
_.unescape([string=''])
```

反向版 `_.escape` 。这个方法转换 HTML 实体 `&` , `<` , `>` , `"` , `'` , 以及 ``` 为对应的字符。

注意: 不会转换其他的 HTML 实体，需要转换可以使用类似 *he* 的第三方库。

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回转换后的字符串

示例

```
_.unescape('fred, barney, &amp; pebbles');
// => 'fred, barney, & pebbles'
```

upperCase

- link
- source
- npm

```
_.upperCase([string=''])
```

转换字符串为空格分割的大写单词

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回大写单词

示例

```
_.upperCase('--foo-bar');
// => 'FOO BAR'

_.upperCase('fooBar');
// => 'FOO BAR'

_.upperCase('__foo_bar__');
// => 'FOO BAR'
```

upperFirst

- link
- source
- npm

```
_.upperFirst([string=''])
```

转换首字母为大写。

参数

1. [string=''] (string)

   要转换的字符串

返回值 (string)

返回转换后的字符串

示例

```
_.upperFirst('fred');
// => 'Fred'

_.upperFirst('FRED');
// => 'FRED'
```

words

- link
- source
- npm

```
_.words([string=''], [pattern])
```

拆分字符串中的词为数组

参数

1. [string=''] (string)

   要处理的字符串

2. [pattern] (RegExp|string)

   匹配模式

返回值 (Array)

然后拆分后的数组

示例

```
_.words('fred, barney, & pebbles');
// => ['fred', 'barney', 'pebbles']

_.words('fred, barney, & pebbles', /[^, ]+/g);
// => ['fred', 'barney', '&', 'pebbles']
```

Util

attempt

- link
- source
- npm

```
_.attempt(func)
```

尝试调用函数，返回结果 或者 错误对象。 任何附加的参数都会在调用时传给函数。

参数

1. func (Function)

   要调用的函数

返回值 (*)

返回函数结果或者错误对象

示例

```
// 避免因为错误的选择器而抛出
var elements = _.attempt(function(selector) {
  return document.querySelectorAll(selector);
}, '>_>');

if (_.isError(elements)) {
  elements = [];
}
```

bindAll

- link
- source
- npm

```
_.bindAll(object, methodNames)
```

绑定对象的方法到对象本身，覆盖已存在的方法。

注意: 这个方法不会设置 "length" 属性到约束的函数。

参数

1. object (Object)

   要绑定的对象

2. methodNames (...(string|string[])

   要绑定的方法名 单独指定或指定在数组中。

返回值 (Object)

返回对象

示例

```
var view = {
  'label': 'docs',
  'onClick': function() {
    console.log('clicked ' + this.label);
  }
};

_.bindAll(view, 'onClick');
jQuery(element).on('click', view.onClick);
// => logs 'clicked docs' when clicked
```

cond

- link
- source
- npm

```
_.cond(pairs)
```

创建一个函数。 这个函数会遍历 `pairs` ，并执行最先返回真值对应的函数，并绑定 `this` 及传入创建函数的参数。

参数

1. pairs (Array)

   判断函数对

返回值 (Function)

返回新的函数

示例

```
var func = _.cond([
  [_.matches({ 'a': 1 }),           _.constant('matches A')],
  [_.conforms({ 'b': _.isNumber }), _.constant('matches B')],
  [_.constant(true),                _.constant('no match')]
]);

func({ 'a': 1, 'b': 2 });
// => 输出：'matches A'

func({ 'a': 0, 'b': 1 });
// => 输出：'matches B'

func({ 'a': '1', 'b': '2' });
// => 输出：'no match'
```

conforms

- link
- source
- npm

```
_.conforms(source)
```

创建一个函数。 这个函数会调用 `source` 的属性名对应的 `predicate` 与传入对象相对应属性名的值进行 `predicate` 处理。 如果都符合返回 `true` ，否则返回 `false`

参数

1. source (Object)

   包含 predicates 属性值的对象

返回值 (Function)

返回新的函数

示例

```
var users = [
  { 'user': 'barney', 'age': 36 },
  { 'user': 'fred',   'age': 40 }
];

_.filter(users, _.conforms({ 'age': _.partial(_.gt, _, 38) }));
// => [{ 'user': 'fred', 'age': 40 }]
```

constant

- link
- source
- npm

```
_.constant(value)
```

创建一个返回 `value` 的函数

参数

1. value (*)

   要返回的值

返回值 (Function)

返回新的函数

示例

```
var object = { 'user': 'fred' };
var getter = _.constant(object);

getter() === object;
// => true
```

flow

- link
- source
- npm

```
_.flow([funcs])
```

创建一个函数。 返回的结果是调用提供函数的结果，`this` 会绑定到创建函数。 每一个连续调用，传入的参数都是前一个函数返回的结果。

参数

1. [funcs] (...(Function|Function[])

   要调用的函数

返回值 (Function)

返回新的函数

示例

```
function square(n) {
  return n * n;
}

var addSquare = _.flow(_.add, square);
addSquare(1, 2);
// => 9
```

flowRight

- link
- source
- npm

```
_.flowRight([funcs])
```

这个方法类似 `_.flow`，除了它调用函数的顺序是从右往左的。

参数

1. [funcs] (...(Function|Function[]))

   要调用的函数

返回值 (Function)

返回新的函数

示例

```
function square(n) {
  return n * n;
}

var addSquare = _.flowRight(square, _.add);
addSquare(1, 2);
// => 9
```

identity

- link
- source
- npm

```
_.identity(value)
```

这个方法返回首个提供的参数

参数

1. value (*)

   任何值

返回值 (*)

返回 value

示例

```
var object = { 'user': 'fred' };

_.identity(object) === object;
// => true
```

iteratee

- link
- source
- npm

```
_.iteratee([func=_.identity])
```

创建一个调用 `func` 的函数。 如果 `func` 是一个属性名，传入包含这个属性名的对象，回调返回对应属性名的值。 如果 `func` 是一个对象，传入的元素有相同的对象属性，回调返回 `true` 。 其他情况返回 `false` 。

参数

1. [func=_.identity] (*)

   转换成 callback 的值

返回值 (Function)

返回 callback.

示例

```
var users = [
  { 'user': 'barney', 'age': 36 },
  { 'user': 'fred',   'age': 40 }
];

// 创建一个自定义 iteratee
_.iteratee = _.wrap(_.iteratee, function(callback, func) {
  var p = /^(\S+)\s*([<>])\s*(\S+)$/.exec(func);
  return !p ? callback(func) : function(object) {
    return (p[2] == '>' ? object[p[1]] > p[3] : object[p[1]] < p[3]
  };
});

_.filter(users, 'age > 36');
// => [{ 'user': 'fred', 'age': 40 }]
```

matches

- link
- source
- npm

```
_.matches(source)
```

创建一个深比较的方法来比较给定的对象和 `source` 对象。 如果给定的对象拥有相同的属性值返回 `true`，否则返回 `false`

注意: 这个方法支持以 `_.isEqual` 的方式比较相同的值。

参数

1. source (Object)

   要匹配的源对象

返回值 (Function)

返回新的函数

示例

```
var users = [
  { 'user': 'barney', 'age': 36, 'active': true },
  { 'user': 'fred',   'age': 40, 'active': false }
];

_.filter(users, _.matches({ 'age': 40, 'active': false }));
// => [{ 'user': 'fred', 'age': 40, 'active': false }]
```

matchesProperty

- link
- source
- npm

```
_.matchesProperty(path, srcValue)
```

创建一个深比较的方法来比较给定对象的 `path` 的值是否是 `srcValue` 。 如果是返回 `true` ，否则返回 `false`

注意: 这个方法支持以 `_.isEqual` 的方式比较相同的值。

参数

1. path (Array|string)

   给定对象的属性路径名

2. srcValue (*)

   要匹配的值

返回值 (Function)

返回新的函数

示例

```
var users = [
  { 'user': 'barney' },
  { 'user': 'fred' }
];

_.find(users, _.matchesProperty('user', 'fred'));
// => { 'user': 'fred' }
```

method

- link
- source
- npm

```
_.method(path, [args])
```

创建一个调用给定对象 `path` 上的函数。 任何附加的参数都会传入这个调用函数中。

参数

1. path (Array|string)

   调用函数所在对象的路径

2. [args] (...*)

   传递给调用函数的参数

返回值 (Function)

返回新的函数

示例

```
var objects = [
  { 'a': { 'b': { 'c': _.constant(2) } } },
  { 'a': { 'b': { 'c': _.constant(1) } } }
];

_.map(objects, _.method('a.b.c'));
// => [2, 1]

_.invokeMap(_.sortBy(objects, _.method(['a', 'b', 'c'])), 'a.b.c')
// => [1, 2]
```

methodOf

- link
- source
- npm

```
_.methodOf(object, [args])
```

反向版 `_.method` 。 这个创建一个函数调用给定 `object` 的 `path` 上的方法， 任何附加的参数都会传入这个调用函数中。

参数

1. object (Object)

   要检索的对象

2. [args] (...*)

   传递给调用函数的参数

返回值 (Function)

返回新的函数

示例

```
var array = _.times(3, _.constant),
    object = { 'a': array, 'b': array, 'c': array };

_.map(['a[2]', 'c[0]'], _.methodOf(object));
// => [2, 0]

_.map([['a', '2'], ['c', '0']], _.methodOf(object));
// => [2, 0]
```

mixin

- link
- source
- npm

```
_.mixin([object=lodash], source, [options])
```

添加来源对象自身的所有可枚举函数属性到目标对象。 如果 `object` 是个函数，那么函数方法将被添加到原型链上。

注意: 使用 `_.runInContext` 来创建原始的 `lodash` 函数来避免修改造成的冲突。

参数

1. [object=lodash] (Function|Object)

   目标对象
2. source (Object)

   来源对象
3. [options] (Object)

   选项对象
4. [options.chain=true] (boolean)

   是否开启链式操作

返回值 (Function|Object)

返回对象

示例

```
function vowels(string) {
  return _.filter(string, function(v) {
    return /[aeiou]/i.test(v);
  });
}

_.mixin({ 'vowels': vowels });
_.vowels('fred');
// => ['e']

_('fred').vowels().value();
// => ['e']

_.mixin({ 'vowels': vowels }, { 'chain': false });
_('fred').vowels();
// => ['e']
```

noConflict

- link
- source
- npm

```
_.noConflict()
```

释放 `_` 为原来的值，并返回一个 `lodash` 的引用

返回值 (Function)

返回 `lodash` 函数

示例

```
var lodash = _.noConflict();
```

noop

- link
- source
- npm

```
_.noop()
```

无论传递什么参数，都返回 `undefined` 。

示例

```
var object = { 'user': 'fred' };

_.noop(object) === undefined;
// => true
```

nthArg

- link
- source
- npm

```
_.nthArg([n=0])
```

创建一个返回第 N 个参数的函数。

参数

1. [n=0] (number)

   要返回参数的索引

返回值 (Function)

返回新的函数

示例

```
var func = _.nthArg(1);

func('a', 'b', 'c');
// => 'b'
```

over

- link
- source
- npm

```
_.over(iteratees)
```

创建一个传入提供的参数的函数并调用 `iteratees` 返回结果的函数。

参数

1. iteratees (...(Function|Function[]))

   要调用的 iteratees

返回值 (Function)

返回新的函数

示例

```
var func = _.over(Math.max, Math.min);

func(1, 2, 3, 4);
// => [4, 1]
```

overEvery

- link
- source
- npm

```
_.overEvery(predicates)
```

创建一个传入提供的参数的函数并调用 `iteratees` 判断是否 全部 都为真值的函数。

参数

1. predicates (...(Function|Function[])

   要调用的 predicates

返回值 (Function)

返回新的函数

示例

```
var func = _.overEvery(Boolean, isFinite);

func('1');
// => true

func(null);
// => false

func(NaN);
// => false
```

overSome

- link
- source
- npm

```
_.overSome(predicates)
```

创建一个传入提供的参数的函数并调用 `iteratees` 判断是否 存在 有真值的函数。

参数

1. predicates (...(Function|Function[])

   要调用的 predicates

返回值 (Function)

返回新的函数

示例

```
var func = _.overSome(Boolean, isFinite);

func('1');
// => true

func(null);
// => true

func(NaN);
// => false
```

property

- link
- source
- npm

```
_.property(path)
```

创建一个返回给定对象的 `path` 的值的函数。

参数

1. path (Array|string)

   要得到值的属性路径

返回值 (Function)

返回新的函数

示例

```
var objects = [
  { 'a': { 'b': { 'c': 2 } } },
  { 'a': { 'b': { 'c': 1 } } }
];

_.map(objects, _.property('a.b.c'));
// => [2, 1]

_.map(_.sortBy(objects, _.property(['a', 'b', 'c'])), 'a.b.c');
// => [1, 2]
```

propertyOf

- link
- source
- npm

```
_.propertyOf(object)
```

反向版 `_.property` 。 这个方法创建的函数返回给定 path 在对象上的值。

参数

1. object (Object)

   要检索的对象

返回值 (Function)

返回新的函数

示例

```
var array = [0, 1, 2],
    object = { 'a': array, 'b': array, 'c': array };

_.map(['a[2]', 'c[0]'], _.propertyOf(object));
// => [2, 0]

_.map([['a', '2'], ['c', '0']], _.propertyOf(object));
// => [2, 0]
```

range

- link
- source
- npm

```
_.range([start=0], end, [step=1])
```

创建一个包含从 `start` 到 `end` ，但不包含 `end` 本身范围数字的数组。 如果 `start` 是负数，而 `end` 或 `step` 没有指定，那么 `step` 从 `-1` 为开始。 如果 `end` 没有指定， `start` 设置为 `0` 。 如果 `end` 小于 `start` ，会创建一个空数组，除非指定了 `step` 。

注意: JavaScript 遵循 IEEE-754 标准处理无法预料的浮点数结果。

参数

1. [start=0] (number)

   开始的范围

2. end (number)

   结束的范围

3. [step=1] (number)

   范围的增量 或者 减量

返回值 (Array)

返回范围内数字组成的新数组

示例

```
_.range(4);
// => [0, 1, 2, 3]

_.range(-4);
// => [0, -1, -2, -3]

_.range(1, 5);
// => [1, 2, 3, 4]

_.range(0, 20, 5);
// => [0, 5, 10, 15]

_.range(0, -4, -1);
// => [0, -1, -2, -3]

_.range(1, 4, 0);
// => [1, 1, 1]

_.range(0);
// => []
```

rangeRight

- link
- source
- npm

```
_.rangeRight([start=0], end, [step=1])
```

这个方法类似 `_.range` ，除了它是降序生成值的。

参数

1. [start=0] (number)

   开始的范围

2. end (number)

   结束的范围

3. [step=1] (number)

   范围的增量 或者 减量

返回值 (Array)

返回范围内数字组成的新数组

示例

```
_.rangeRight(4);
// => [3, 2, 1, 0]

_.rangeRight(-4);
// => [-3, -2, -1, 0]

_.rangeRight(1, 5);
// => [4, 3, 2, 1]

_.rangeRight(0, 20, 5);
// => [15, 10, 5, 0]

_.rangeRight(0, -4, -1);
// => [-3, -2, -1, 0]

_.rangeRight(1, 4, 0);
// => [1, 1, 1]

_.rangeRight(0);
// => []
```

runInContext

- link
- source
- npm

```
_.runInContext([context=root])
```

创建一个给定上下文对象的原始的 `lodash` 函数。

参数

1. [context=root] (Object)

   上下文对象

返回值 (Function)

返回新的 `lodash` 对象

示例

```
_.mixin({ 'foo': _.constant('foo') });

var lodash = _.runInContext();
lodash.mixin({ 'bar': lodash.constant('bar') });

_.isFunction(_.foo);
// => true
_.isFunction(_.bar);
// => false

lodash.isFunction(lodash.foo);
// => false
lodash.isFunction(lodash.bar);
// => true

// 使用 `context` 模拟 `Date#getTime` 调用 `_.now`
var mock = _.runInContext({
  'Date': function() {
    return { 'getTime': getTimeMock };
  }
});

// 或者在 Node.js 中创建一个更高级的 `defer`
var defer = _.runInContext({ 'setTimeout': setImmediate }).defer;
```

times

- link
- source
- npm

```
_.times(n, [iteratee=_.identity])
```

调用 iteratee N 次，每次调用返回的结果存入到数组中。 iteratee 会传入1个参数：(index)。

参数

1. n (number)

   要调用 `iteratee` 的次数

2. [iteratee=_.identity] (Function)

   这个函数会处理每一个元素

返回值 (Array)

返回调用结果的数组

示例

```
_.times(3, String);
// => ['0', '1', '2']

 _.times(4, _.constant(true));
// => [true, true, true, true]
```

toPath

- link
- source
- npm

```
_.toPath(value)
```

创建 `value` 为属性路径的数组

参数

1. value (*)

   要转换的值

返回值 (Array)

返回包含属性路径的数组

示例

```
_.toPath('a.b.c');
// => ['a', 'b', 'c']

_.toPath('a[0].b.c');
// => ['a', '0', 'b', 'c']

var path = ['a', 'b', 'c'],
    newPath = _.toPath(path);

console.log(newPath);
// => ['a', 'b', 'c']

console.log(path === newPath);
// => false
```

uniqueId

- link
- source
- npm

```
_.uniqueId([prefix])
```

创建唯一ID。 如果提供了 `prefix` ，会被添加到ID前缀上。

参数

1. [prefix] (string)

   要添加到ID前缀的值

返回值 (string)

返回唯一ID

示例

```
_.uniqueId('contact_');
// => 'contact_104'

_.uniqueId();
// => '105'
```